实战 Groovy 系列

wizardforcel

Published with GitBook



目錄

实战 Groovy 系列

> 来源：实战 Groovy 系列

Groovy 是运行在 Java 平台上的现代编程语言。它将提供与现有 Java 代码的无缝集成，同时引入闭包和元编程等出色的新功能。简言之，Groovy 是 21 世纪根据 Java 语言的需要编写的。

把任意一个新工具集成到开发工具包中的关键是：知道何时使用以及何时不应使用该工具。Groovy 可以提供强大的功能，但是必须正确地应用到适当的场景中。为此，实战 Groovy 系列将探究 Groovy 的实际应用，帮助您了解何时及如何成功应用它们。

实战 Groovy: SwingBuilder 和 Twitter API，第 2 部分

使用 *HTTP* 基本身份验证和 *ConfigSlurper*

在本期 *实战 Groovy* 文章中，Scott Davis 将继续构建 第 1 部分 中的 Groovy Twitter 客户机：Gwitter。这次，他将解决 HTTP Basic 身份验证问题，并使用 Groovy 的 `ConfigSlurper` 读入配置设置。

在 “SwingBuilder 和 Twitter API，第 1 部分” 中，我使用 Groovy 的 `SwingBuilder` 建立了一个简单 Swing GUI ，从而创建了一个 Twitter 客户机（名称为 Gwitter）。我展示了通用的 Twitter Search API，它可以方便地绕过对身份验证的需求。但是 Twitter 的典型使用模式是获取所关注的人（在 Twitter 中为朋友）的 tweet，并发表自己的 tweet。要能够读取朋友的 tweet，Twitter 必须首先知道您是谁。因此，您必须能够使用 Twitter API 以编程的方式登录。

关于本系列

Groovy 是一种新兴的运行于 Java™ 平台之上的编程语言。它提供与已有 Java 代码的无缝集成，并引入了一些强大的新特性，比如闭包和元编程。简单来讲，Groovy 就是在 21 世纪 Java 语言的的效果。

将任何新工具整合进开发工具包中的关键是知道何时使用它以及何时保留它。Groovy 有时非常强大，但仅仅适用于一些适当的场景。因此，实战 *Groovy* 系列将探索 Groovy 的实际使用，从而帮助您了解如何成功地应用它们。

在本文中，您将向 Gwitter 添加身份验证功能，并使 Gwitter 能够请求和解析您的朋友时间轴。与上次一样，您首先将了解基本的命令行概念，在确认它能独立工作之后，将它整合到 Gwitter 中 — 并对 Gwitter UI 进行一些增强。要获取本文的完整源代码，请参见 下载 小节。

Twitter 中的身份验证

Twitter 支持两种身份验证方法：OAuth 和 HTTP Basic 身份验证。Twitter 开发团队在文档中明确表明建议您首选 OAuth 机制（参见 参考资料）。但他们也承认 “针对桌面客户机的传统 OAuth 流程有时非常麻烦”。因此，我将使用 Groovy 中最为简单的 HTTP Basic 身份验证方式，相信大家也不会感到奇怪。

您自己的 Twitter 帐户

要让本文中的示例正常运行，您需要提供自己的 Twitter 用户名和密码。如果还没有帐户，请访问 https://twitter.com/signup。

HTTP Basic 身份验证的流行归因于它的的实例非常简单（参见 参考资料）。您只需要使用冒号来连接用户名和密码，然后对结果执行 Base64 编码。可以使用最朴实的语言来描述它的安全性：它可以防止纯文本凭证通过线路传递出去。但是，与 Web 开发相关的任何编程语言都可以使用 Base64 编码和解码。

在 Groovy中，Base64 编码的 Twitter 用户名和密码如下所示：

```
def authString = "username:password".getBytes().encodeBase64().toSt
```

创建身份验证字符串仅仅是整个问题中的一小部分。为了能够读取朋友的 tweet，您需要向 http://twitter.com/statuses/friends_timeline.atom 发出一个 REST 式请求，并在 HTTP GET 请求的 `Authorization` 头部中传入 Base6 编码的字符串（参见 参考资料）。

创建一个名称为 friends.groovy 文件。添加如清单 1 所示的代码：

1. 以 Atom 文档的形式请求朋友时间轴。
2. 使用 `XmlSlurper` 解析它。
3. 向控制台打印输出结果。

（要回顾如何使用 Groovy `XmlSlurper` 来解析 XML，请阅读 “实战 Groovy：构建和解析 XML”。）

清单 1. 在 Twitter 中请求朋友时间轴

```
def addr = "http://twitter.com/statuses/friends_timeline.atom"
def authString = "username:password".getBytes().encodeBase64().toSt
def conn = addr.toURL().openConnection()
conn.setRequestProperty("Authorization", "Basic ${authString}")
if(conn.responseCode == 200){
  def feed = new XmlSlurper().parseText(conn.content.text)
  feed.entry.each{entry->
    println entry.author.name
    println entry.title
    println "-"*20
  }
}else{
  println "Something bad happened."
  println "${conn.responseCode}: ${conn.responseMessage}"
}
```

在命令行中输入 `groovy friends` 。输出应该类似于清单 2。当然，您的输出将根据传递给 Twitter 的凭证以及所关注的朋友而有所不同。

清单 2. friends.groovy 的 Twitter 输出

```
--------------------
Scott Davis
scottdavis99: @neal4d Is the Bishop's Arms *diagonally* adjacent
to your hotel? Or is it two doors over and one door up?
--------------------
Neal Ford
neal4d: At the Bishop's Arms, the pub adjacent our hotel, with
a shocking number of single-malt scotches. I need to spend some
quality time here.
--------------------
```

cURL

`cURL` 是一个用于发起 HTTP GET、POST、PUT 和 DELETE 请求的命令行实用工具。它是 UNIX®、Linux® 和 Mac OS X 系统上的标准工具，并且可以下载到 Windows® 中（参见 参考资料）。（有关使用 `cURL` 与 REST 式 Web 服务交互的更多信息，请阅读 “精通 Grails：RESTful Grails”。）

要掌握原始 Atom 输出，可以在命令行中发起一个 cURL 请求。清单 3 给出了一个示例：

清单 3. 使用 `cURL` 从 Twitter 获取原始 Atom

```
$ curl -u scottdavis99:password
    http://twitter.com/statuses/friends_timeline.atom

<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <title>Twitter / scottdavis99 with friends</title>
  <updated>2009-11-03T22:15:45+00:00</updated>

  <entry>
    <title>scottdavis99: @neal4d Is the Bishop's Arms *diagonally*
      adjacent to your hotel? Or is it two doors over and one door
    </title>
    <id>tag:twitter.com,2007:
       http://twitter.com/scottdavis99/statuses/5402041984</id>
    <published>2009-11-03T21:23:43+00:00</published>
    <updated>2009-11-03T21:23:43+00:00</updated>
    <author>
      <name>Scott Davis</name>
      <uri>http://thirstyhead.com</uri>
    </author>
  </entry>

  <entry />
  <entry />
  <entry />
</feed>
```

这样，现在您已经确定可以使用 Basic 身份验证来发起 HTTP GET 请求并解析结果了。但是，源代码中硬编码的用户名和密码会生成一个红色标记。较好的方法是将身份验证信息重构到外部属性文件中。

使用 Groovy 的 `ConfigSlurper` 读取 Java 属性

在 Java 应用程序中存储外部信息的一种最常用的方法是使用属性文件。属性文件是存储在纯文本中的简单键值对。属性文件是平面文件 — 它们并未提供任何形式的键嵌套和分组 — 但是您可以通过创建具有相同前缀的带点名称来对您的键实现虚假嵌套。

创建一个名称为 config.properties 的文件，并添加如清单 4 所示的内容：

清单 4. 一个简单的 Java 属性文件

```
login.username=fred
login.password=wordpass
```

读取属性文件的一种方法是使用 `java.util.Properties` 类。创建一个名称为 PropertiesTest.groovy 的文件并编写如清单 5 所示的单元测试：

清单 **5.** 读取 **Java** 属性文件的简单单元测试

```
class PropertiesTest extends GroovyTestCase{
  void testReadingProperties(){
    Properties properties = new Properties();
    try{
      properties.load(new FileInputStream("config.properties"));
      assertEquals("fred", properties.getProperty("login.username")
      assertEquals("wordpass", properties.getProperty("login.passwo
    }catch (IOException e) {
      fail(e.getMessage());
    }
  }
}
```

要运行单元测试，在命令行中键入 `groovy PropertiesTest` 。每个测试方法应该都有一个点，并且在运行完成后会显示 `OK (1 test)` 。

您已经了解了使用 `XmlSlurper` 来剖析 XML 是多么地简单。Groovy 提供了一个类似的类 — `groovy.util.ConfigSlurper` — 用于读入属性文件。`ConfigSlurper` 使读取 Java 属性文件变得非常简单。

向 PropertiesTest.groovy 添加一个新测试，如清单 6 所示：

清单 **6.** 通过 **`ConfigSlurper`** 读取 **Java** 属性文件

```
void testReadingPropertiesWithConfigSlurper(){
  Properties properties = new Properties();
  properties.load(new FileInputStream("config.properties"));
  def config = new ConfigSlurper().parse(properties);
  assertEquals "fred", config.login.username
  assertEquals "wordpass", config.login.password
}
```

如您所见，`ConfigSlurper` 允许您使用相同的点表示来遍历 Java 属性（`XmlSlurper` 支持的 XML 方式）。`ConfigSlurper GPath` 语法要比清单 5 的测试中所使用的 `properties.getProperty("login.username")` 简单很多。

但是，`ConfigSlurper` 的益处已经超越了语法的可读性。您可以使用一组非常类似于 Groovy 的嵌套配置块在配置文件中存储您的值。

使用 `ConfigSlurper` 读取 Groovy 配置

几乎可以说 Groovy 配置文件是用于存储配置设置的域相关语言 (DSL)。您可以将相似的设置分组到一个块中，并且可以将一些块嵌入到任意深度。这种嵌套方式既可以减少重复键入，又可以确保相关设计的清晰性。

创建一个名称为 config.groovy 的文件并添加如清单 7 所示的代码 ：

清单 **7. Groovy** 配置文件

```
login{
  username = "fred"
  password = "wordpass"
}
```

向 PropertiesTest.groovy 再添加一个测试，用于读入 Groovy 配置文件，如清单 8 所示：

清单 **8.** 使用 `ConfigSlurper` 读取 **Groovy** 配置文件

```
void testConfigSlurper(){
  def config = new ConfigSlurper().parse(new File("config.groovy")
  assertEquals "fred", config.login.username
  assertEquals "wordpass", config.login.password
}
```

由于过载的解析方法可以接受一个简单的 `String` ，因此在一个单元测试中模拟配置设置很繁琐。Groovy 的三重引用可以很好地解决此问题，如清单 9 所示：

清单 **9.** 创建模拟设置

```
void testMockConfig(){
  def mockConfig = """
    smtp{
      server = "localhost"
      port = 25
      auth{
        user = "testuser"
        password = "testpass"
      }
    }
  """

  def config = new ConfigSlurper().parse(mockConfig)
  assertEquals "testuser", config.smtp.auth.user
}
```

结合使用 Groovy 配置文件（而不是 Java 属性文件）与 `ConfigSlurper` 将提供更加具体的收益。您可以根据预定义的环境，比如 `development`、`production` 和 `testing`，来切换各值。这可以极大的提高可测试性。

通过 `ConfigSlurper` 来使用环境

如果在 Groovy 配置文件中创建了一个特别命名的块 — `environments`，那么可以选择性地覆盖配置设置。如果您是一名 Grails 用户，则应该清楚它在 grails-app/conf/DataSource.groovy 中的工作原理，如清单 10 所示：

清单 10. 使用 `ConfigSlurper` 管理数据库连接的 Grails

```
dataSource {
   pooled = true
   driverClassName = "org.hsqldb.jdbcDriver"
   username = "sa"
   password = ""
}

// environment specific settings
environments {
   development {
      dataSource {
         dbCreate = "create-drop" // one of 'create', 'create-drop
         url = "jdbc:hsqldb:mem:devDB"
      }
   }
   test {
      dataSource {
         dbCreate = "update"
         url = "jdbc:hsqldb:mem:testDb"
      }
   }
   production {
      dataSource {
         dbCreate = "update"
         url = "jdbc:hsqldb:file:prodDb;shutdown=true"
      }
   }
}
```

在清单 10 中，`ConfigSlurper` 传回的最后一个 `dataSource` 配置对象是顶部 `dataSource` 块中的全局值与 `development`、`test` 或 `production` 块中的环境相关设置的组合。您在 `environments` 块中设置的值将覆盖全局值。

最外层的块必须被命名为 `environments`，但您可以将内层块命名为任何名称。在 config.groovy 中添加如清单 11 所示的代码，然后在 PropertiesTest.groovy 中测试它：

清单 11. 在 Groovy 配置文件中使用环境

```
//config.groovy
login{
  username = "fred"
  password = "wordpass"
}

environments{
  qa{
    login{
      username = "testuser"
      password = "testpass"
    }
  }
}

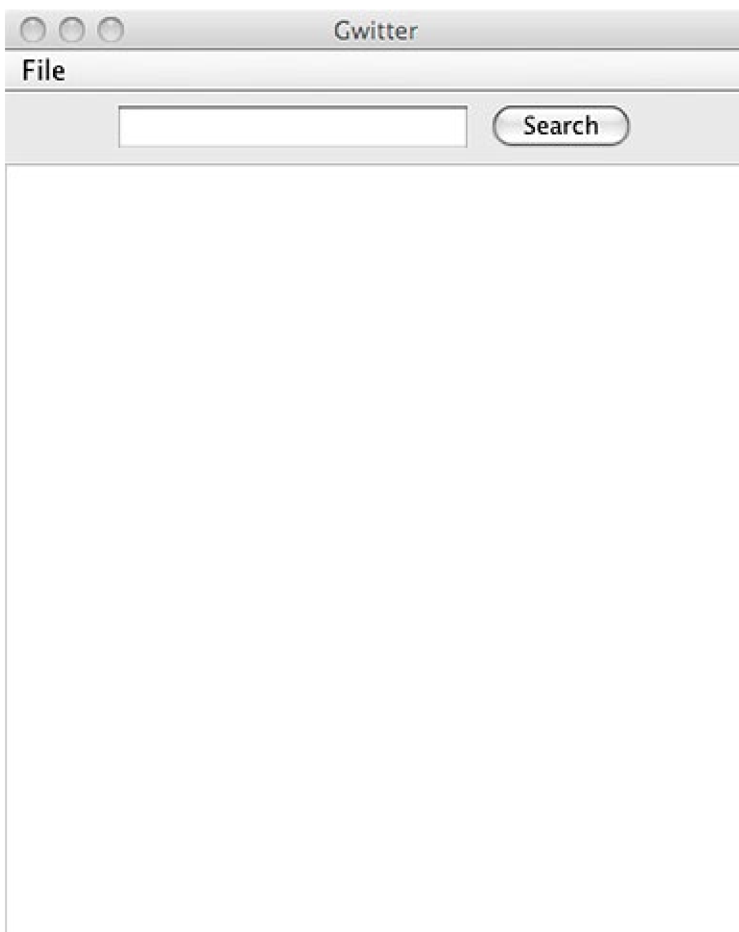
//PropertiesTest.groovy
void testWithEnvironment(){
  def config = new ConfigSlurper("qa").parse(new File("config.groov
  assertEquals "testuser", config.login.username
  assertEquals "testpass", config.login.password
}
```

现在，您已经具备了一定的 `ConfigSlurper` 经验。接下来，结合它与之前运行的朋友时间轴示例，将它们添加到 Gwitter Swing 应用程序中。首先，您需要切换为分页界面，以便于显示新的信息。

向 Gwitter 添加一个 `TabbedPane`

在 第 1 部分 结束时，您创建了一个单窗格的应用程序，如图 1 所示：

图 1. 单窗格 Gwitter 应用程序



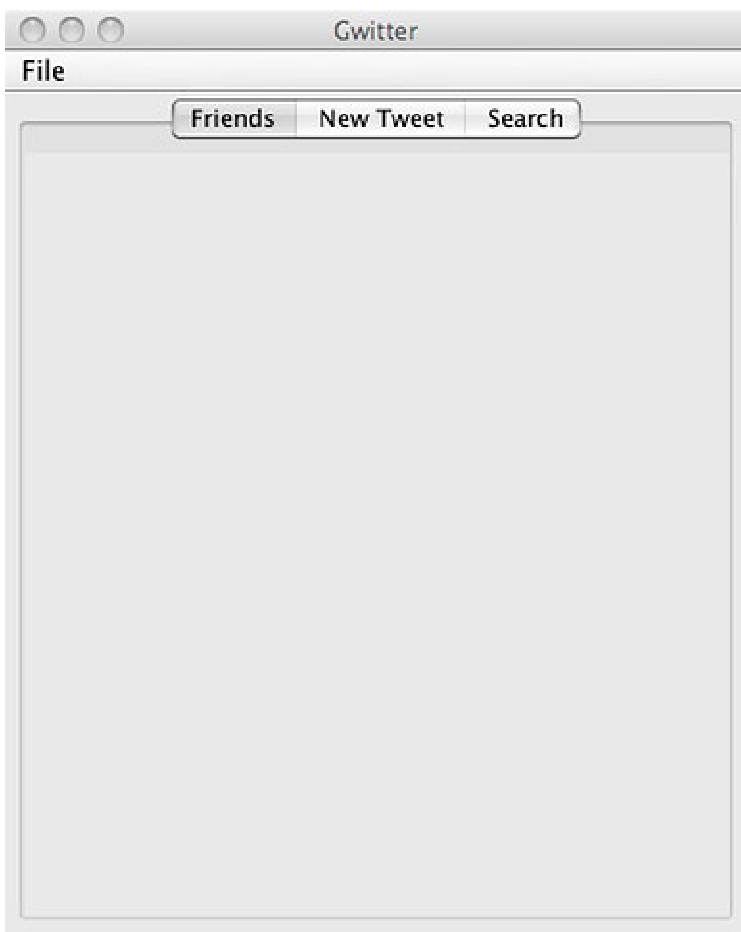
要显示朋友时间轴，您需要切换为分页视图。应该还记得 Groovy 的 `SwingBuilder` 可以脱离典型的 Java 语法 `JTabbedPane tabpane = new JTabbedPane();` ，而只需向 `SwingBuilder.frame` 添加一个嵌套的 `tabbedPane` 闭包。（类似于 Groovy 配置文件和 `ConfigBuilder` ，不是吗？）向 Gwitter 添加一个新的 `tabbedPane` ，如清单 12 所示：

清单 12. 向 Gwitter 添加一个 `tabbedPane`

```
swingBuilder.frame(title:"Gwitter",
                   defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                   size:[400,500],
                   show:true) {
  customMenuBar()
  tabbedPane{
    panel(title:"Friends")
    panel(title:"New Tweet")
    panel(title:"Search"){
      searchPanel()
      resultsPanel()
    }
  }
}
```

可以预见，`title` 属性将显示在选项卡中。在命令行中键入 `groovy Gwitter` ，应该能够看到如图 2 所示的选项卡：

图 **2.** 带有新分页界面的 **Gwitter**



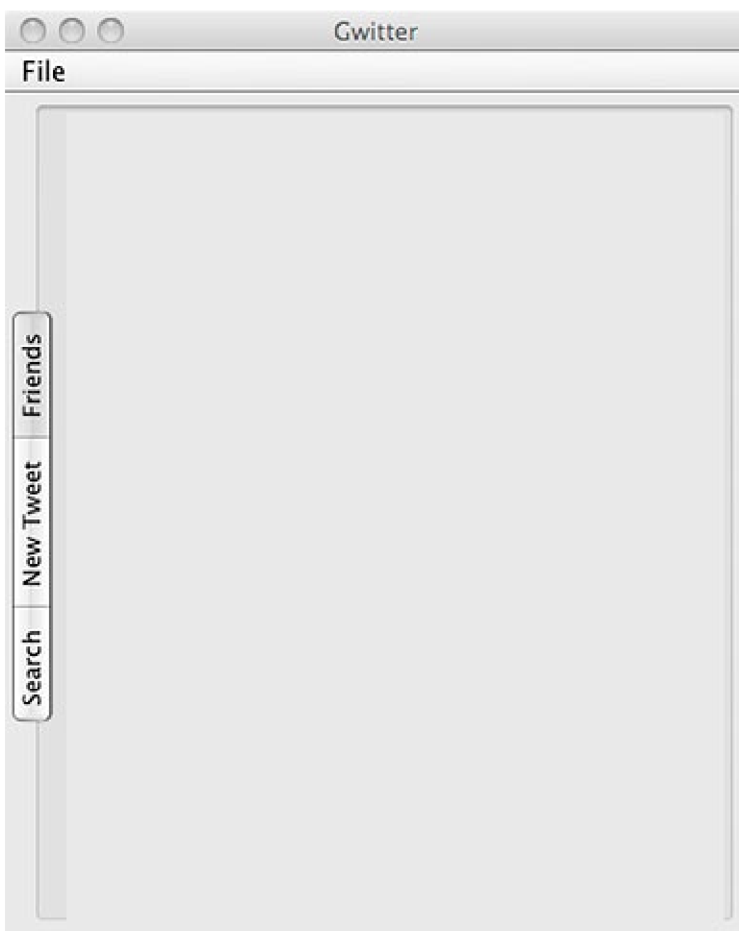
再次强调，`SwingBuilder` 仅仅在传统 Java Swing 上添加了一层语法。所有原 Java 技巧仍然适用于 Groovy。举例来说，无需使用在一行中实例化 `JTabbedPane` 并在第二行中调用 `setTabPlacement()` 这样的 setter 方法，您可以在一行中完成这两个任务，如清单 13 所示：

清单 **13.** 在分页面板的其余部分放置些选项卡

```
swingBuilder.frame(title:"Gwitter",
                   defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                   size:[400,500],
                   show:true) {
  customMenuBar()
  tabbedPane(tabPlacement:JTabbedPane.LEFT){
    panel(title:"Friends")
    panel(title:"New Tweet")
    panel(title:"Search"){
      searchPanel()
      resultsPanel()
    }
  }
}
```

如您所料，这将在面板左侧对齐选项卡，如图 3 所示：

图 **3.** 在分页窗格左侧对齐选项卡



运行应用程序并确认其他 `tabPlacement` 可正常工作之后，删除将选项卡返回到面板顶部的默认位置的设置。

填充 Friends 选项卡

现在，您将使用 Twitter API 中的实时数据来填充空的 Friends 选项卡

首先，在与 Gwitter.groovy 相同的目录中创建一个 gwitterConfig.groovy 文件。添加如清单 14 所示的代码（记住要将凭证更改为您的 Twitter 用户名和密码）：

清单 **14. gwitterConfig.groovy** 文件

```
login{
  username = "username"
  password = "password"
}
```

接下来，在相同目录中创建一个 FriendsTimeline.groovy 文件。添加如清单 15 所示的代码：

清单 **15. FriendsTimeline.groovy** 文件

```
class FriendsTimeline{
  static final String addr = "http://twitter.com/statuses/friends_t

  static Object[] refresh(){
    def results = []

    def configFile = new File("gwitterConfig.groovy")
    if(configFile.exists()){
      def config = new ConfigSlurper().parse(configFile.text)

      //NOTE: this should be a single line in your code
      def authString = "${config.login.username}:${config.login.pas
        .getBytes().encodeBase64().toString()
      def conn = addr.toURL().openConnection()
      conn.setRequestProperty("Authorization", "Basic ${authString]
      if(conn.responseCode == 200){
        def feed = new XmlSlurper().parseText(conn.content.text)
        feed.entry.each{entry->
          def tweet = new Tweet()
          tweet.author = entry.author.name
          tweet.published = entry.published
          tweet.content = entry.title
          results << tweet
        }
      }else{
        println "Something bad happened."
        println "${conn.responseCode}: ${conn.responseMessage}"
      }
    }else{
      println "Cannot find ${configFile.name}."
    }

    return results as Object[]
  }
}
```

清单 15 几乎与 Search.groovy 中的代码相同（参见 第 1 部分）。区别在于，FriendsTimeline.groovy 中使用 `ConfigSlurper` 并使用 HTTP Basic 身份验证来读取 gwitterConfig.groovy 中的值。在 Search.groovy 中，您发起了一个简单的无需身份验证的 HTTP GET 请求。

您将 `FriendsTimeline` 类整合到了主 `Gwitter` 类中，这与 第 1 部分中 整合 `Search` 类的方法如出一辙。首先，为 `friendsList` 添加一个字段，如清单 16 所示：

清单 16. 向 Gwitter 添加一个 `friendsList` 字段

```
class Gwitter{
  def searchField
  def resultsList
  def friendsList

  // snip...
}
```

接下来，为 Refresh 按钮以及 `FriendsTimeline.refresh()` 方法调用的结果添加一个面板，如清单 17 所示：

清单 **17.** 向 **Gwitter** 添加两个新面板

```
class Gwitter{
  def searchField
  def resultsList
  def friendsList

  void show(){
    // snip...

    def friendsRefreshPanel = {
      swingBuilder.panel(constraints: BorderLayout.NORTH){
        button(text:"Refresh", actionPerformed:{
          doOutside{
            friendsList.listData = FriendsTimeline.refresh()
          }
        } )
      }
    }

    def friendsPanel = {
      swingBuilder.scrollPane(constraints: BorderLayout.CENTER){
        friendsList = list(fixedCellWidth: 380,
                           fixedCellHeight: 75,
                           cellRenderer:new StripeRenderer())
      }
    }

  }
}
```

最后，在 Friends 选项卡中呈现 `friendsRefreshPanel` 和 `friendsPanel`，如清单 18 所示：

清单 **18.** 在 **Friends** 选项卡中呈现两个面板

```
swingBuilder.frame(title:"Gwitter",
                   defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                   size:[400,500],
                   show:true) {
  customMenuBar()
  tabbedPane{
    panel(title:"Friends"){
      friendsRefreshPanel()
      friendsPanel()
    }
    panel(title:"New Tweet")
    panel(title:"Search"){
      searchPanel()
      resultsPanel()
    }
  }
}
```

在命令行中键入 `groovy Gwitter`，并单击 Refresh 按钮。结果应该如图 4 所示：

图 **4.** 显示朋友时间轴的 **Gwitter**



结束语

在本文中，您学习如何使用 Basic 身份验证发起 HTTP GET 请求。您还了解了如何创建大多数 Groovy 配置文件和 `groovy.util.ConfigSlurper` 。最后，您向 Gwitter 添加了一个分页界面，用于显示 REST 式 Twitter API 调用的结果， 以便于返回朋友时间轴。

下一期，您将发起 HTTP POST 请求来添加自己的 tweet。在此过程中，您还将了解 `JTextArea` 字段以及如何使用 `DocumentSizeFilter` 来限制输入的具体字符数量（比如说 140）。最后，我希望您可以掌握 Groovy 的大量实际应用。

下载

描述	名字	大小
源代码	j-pg11179.zip	21KB

实战 Groovy: SwingBuilder 和 Twitter API，第 1 部分

构建基于 *Swing* 的 *GUI* 从未如此简便

在这一期 实战 *Groovy* 中，Scott Davis 要讨论一个令大多数服务器端 Java™ 开发人员畏惧的主题：Swing。Groovy 的 `SwingBuilder` 可以让这个强大但复杂的 GUI 框架使用起来简单一些。

我最近会见了 Ted Neward，他是 IBM developerWorks 文章系列 面向 *Java* 开发人员的 *Scala* 指南 的作者（见 参考资料）。我们讨论了他在这个系列中构建的一个有意思的 Twitter 库，Scitter (Scala + Twitter)。Scitter 的重点在于 Scala 的 Web 服务和 XML 解析功能，Ted 承认他不太关心为这个 API 提供前端。当然，这启发我考虑用 Groovy 编写一个 Twitter GUI 会怎么样？*Gwitter* (Groovy + Twitter) 是个不错的名字吧？

在本文中我不打算讨论 Scala 和 Groovy 的集成，尽管在这两种语言之间确实有许多协作的可能性。相反，我要讨论 Java 领域中常常被 Java 开发人员忽视的一个主题：Swing。但是，在此之前，我先谈谈 Groovy 的 `XmlSlurper` 如何简化 Twitter 的 Atom feed。

Twitter Search API

看一下 Twitter Search API 的在线文档（见 参考资料）。文档表明可以通过发出简单的 HTTP GET 请求搜索 Twitter。查询通过查询字符串中的 `q` 参数传递，结果以 Atom（一种 XML 联合格式）或 JavaScript Object Notation (JSON) 的形式返回。因此，要想以 Atom 的形式得到所有提到 *thirstyhead* 的条目，需要发出下面这样的 HTTP GET 请求：`http://search.twitter.com/search.atom?q=thirstyhead`。

如清单 1 所示，返回的结果是嵌套在 `<feed>` 元素中的一系列 `<entry>` 元素：

清单 1. Twitter 搜索 Atom 结果

```
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <entry>
    <title>thirstyhead: New series from Andrew Glover: Java Develop
           http://bit.ly/bJX5i</title>
    <content type="html">thirstyhead: New series from Andrew Glover
                         Development 2.0 http://bit.ly/bJX5i</conte
    <id>tag:twitter.com,2007:
        http://twitter.com/thirstyhead/statuses/3419507135</id>
    <published>2009-08-20T02:54:54+00:00</published>
    <updated>2009-08-20T02:54:54+00:00</updated>
    <link type="text/html" rel="alternate"
          href="http://twitter.com/thirstyhead/statuses/3419507135'
    <link type="image/jpeg" rel="image"
          href="http://s3.amazonaws.com/twitter_production/profile_
          73550313/flame_normal.jpg"/>
    <author>
      <name>ThirstyHead.com</name>
      <uri>http://www.thirstyhead.com</uri>
    </author>
  </entry>

  <entry>...</entry>
  <entry>...</entry>
  <!-- snip -->
</feed>
```

在“实战 Groovy：构建和解析 XML”中，可以看到很容易使用 Groovy 的 `XmlSlurper` 处理 XML 结果。既然了解了这些结果的形式，就来创建一个名为 searchCli.groovy 的文件，见清单 2：

清单 **2.** 解析 **Atom** 结果的 **Groovy** 脚本

```
if(args){
 def username = args[0]
 def addr = "http://search.twitter.com/search.atom?q=${username}"
 def feed = new XmlSlurper().parse(addr)
 feed.entry.each{
   println it.author.name
   println it.published
   println it.title
   println "-"*20
 }
}else{
 println "USAGE: groovy searchCli <query>"
}
```

在命令行上输入 `groovy searchCli thirstyhead`，就会显示简洁的 Atom 结果，见清单 3：

清单 **3.** 运行 **searchCli.groovy** 脚本

```
$ groovy searchCli thirstyhead

thirstyhead (ThirstyHead.com)
2009-08-20T02:54:54Z
New series from Andrew Glover:
Java Development 2.0 http://bit.ly/bJX5i
--------------------
kung_foo (kung_foo)
2009-08-18T12:33:32Z
ThirstyHead interviews Venkat Subramaniam:
http://blip.tv/file/2484840 "Groovy and Scala are good friends..."
(via @mittie). very good.

//snip
```

创建最初的 Gwitter 类

Groovy 脚本很适合编写非正式的实用程序和证实概念，但是编写 Groovy 类也不太困难。另外，可以编译 Groovy 类并从 Java 代码调用它们。

例如，可以编写清单 4 所示的 Tweet.groovy：

清单 **4. Tweet.groovy**

```
class Tweet{
  String content
  String published
  String author

  String toString(){
    return "${author}: ${content}"
  }
}
```

这是一个 Plain Old Groovy Object (POGO)，是非常复杂的 Plain Old Java Object (POJO) 的替代品。

现在，把 清单 2 中的搜索脚本转换为 Search.groovy，见清单 5：

清单 **5. Search.groovy**

```
class Search{
  static final String addr = "http://search.twitter.com/search.ator

  static Object[] byKeyword(String query){
    def results = []
    def feed = new XmlSlurper().parse(addr + query)
    feed.entry.each{entry->
      def tweet = new Tweet()
      tweet.author = entry.author.name
      tweet.published = entry.published
      tweet.content = entry.title
      results << tweet
    }
    return results as Object[]
  }
}
```

通常情况下，我会让结果保持 `java.util.ArrayList` 的形式。但是，本文后面使用的 `javax.swing.JList` 需要一个 `Object[]`，所以这里提前做一些准备。

注意，我在 Search.groovy 中去掉了 `main()` 方法。现在如何与这个类交互呢？当然可以通过单元测试！创建 SearchTest.groovy，见清单 6：

清单 **6. SearchTest.groovy**

```
class SearchTest extends GroovyTestCase{
  void testSearchByKeyword(){
    def results = Search.byKeyword("thirstyhead")
    results.each{
      assertTrue it.content.toLowerCase().contains("thirstyhead")
                 it.author.toLowerCase().contains("thirstyhead")
    }
  }
}
```

如果在命令提示上输入 `groovy SearchTest`，然后看到 `OK (1 test)`（见清单 7），就说明已经成功地把搜索脚本转换为可重用的类了：

清单 **7.** 成功测试的运行结果

```
$ groovy SearchTest
.
Time: 4.64

OK (1 test)
```

现在底层基础结构已经就位了，下一步是开始为它提供漂亮的前端。

`SwingBuilder` 简介

Swing 是一个极其强大的 GUI 工具集。但糟糕的是，有时候其复杂性会影响开发人员挥发它的能力。如果您刚接触 Swing，会觉得像是在学习开波音 747，而您实际上只需要开单引擎的 Cessna 或滑翔机。

Groovy 的 `SwingBuilder` 并不能降低各种任务内在的复杂性，比如选择适当的 `LayoutManager` 或处理线程问题。它降低的是语法复杂性。Groovy 的命名参数/变量参数构造器非常适合需要实例化的各种 `JComponent` ，然后马上可以为它们配置一系列设置器。（关于 `SwingBuilder` 的更多信息，请参见 参考资料）。

但是，同样有价值的是 Groovy 对闭包的使用。对于 Swing，我长期关注的问题是自然的层次结构似乎在实现细节中消失了。在 Java 代码中，会得到一组相互脱节的组件，看不出哪个组件属于哪个组件。可以以任意次序声明 `JFrame` 、 `JPanel` 和 `JLabel` 。在代码中，它们看起来是平等的；但是，实际上 `JFrame` 包含 `JPanel` ， `JPanel` 进而包含 `JLabel` 。清单 8 给出一个示例：

清单 **8. HelloJavaSwing.java**

```
import javax.swing.*;

public class HelloJavaSwing {
  public static void main(String[] args) {
    JPanel panel = new JPanel();
    JLabel label = new JLabel("Hello Java Swing");

    JFrame frame = new JFrame("Hello Java Swing");
    panel.add(label);
    frame.add(panel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(200,300);
    frame.setVisible(true);
  }
}
```

编译这段代码（`javac HelloJavaSwing.java`）并运行它（`java HelloJava`），应该会显示图 1 所示的应用程序：

图 1. **`HelloJavaSwing`**



清单 9 给出用 Groovy 编写的同一个应用程序。可以看到 `SwingBuilder` 使用了闭包，这让我们可以清晰地看出拥有关系链。

清单 **9. HelloGroovySwing.groovy**

```
import groovy.swing.SwingBuilder
import javax.swing.*

def swingBuilder = new SwingBuilder()
swingBuilder.frame(title:"Hello Groovy Swing",
                   defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                   size:[200,300],
                   show:true) {
  panel(){
    label("Hello Groovy Swing")
  }
}
```

输入 `groovy HelloGroovySwing` 会看到图 2 所示的应用程序：

图 **2.** `HelloGroovySwing`



注意，在 清单 9 中，所有组件名去掉了开头的 `J`，方法名中也去掉了多余的 `get` 和 `set`。接下来，注意 `frame` 的命名参数构造器。在幕后，Groovy 调用无参数构造器，然后调用设置器方法，这与前面的 Java 示例没有区别。但是，设置器方法都集中在构造器中，代码更简洁了，去掉 `set` 前缀和末尾的圆括号也大大减少了视觉干扰。

如果您不了解 Swing，这段代码看起来可能仍然比较复杂。但是，如果您具备哪怕最粗浅的 Swing 经验，就可以看出它具有 Swing 的特征：干净、清晰和高效。

正如在前一节中所做的，通过脚本了解概念，然后把脚本转换为类。创建文件 Gwitter.groovy，见清单 10。这是 Groovy + Twitter 客户机 UI 的起点。

清单 **10. Gwitter UI** 的骨架

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()
    swingBuilder.frame(title:"Gwitter",
                       defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                       size:[400,500],
                       show:true) {
    }
  }
}
```

输入 `groovy Gwitter` ，确认会出现空的框架。如果一切正常，下一步是在应用程序中添加一个简单的菜单。

添加菜单栏

在 Swing 中创建菜单提供另一个具有自然层次结构的组件示例。创建一个 `JMenuBar` ，它包含一个或多个 `JMenu` ， `JMenu` 进而包含一个或多个 `JMenuItem` 。

为了创建包含 `Exit` 菜单项的 `File` 菜单，在 Gwitter.groovy 中添加清单 11 中的代码：

清单 11. 在 Gwitter 中添加 `File` 菜单

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()

    def customMenuBar = {
      swingBuilder.menuBar{
        menu(text: "File", mnemonic: 'F') {
          menuItem(text: "Exit", mnemonic: 'X', actionPerformed: {
        }
      }
    }

    swingBuilder.frame(title:"Gwitter",
                       defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                       size:[400,500],
                       show:true) {
      customMenuBar()
    }
  }
}
```

请注意 `customMenuBar` 闭包的嵌套层次结构。为了便于阅读，这里添加了换行和缩进，但是同样很容易在同一行中定义它。定义这个闭包之后，在 `frame` 闭包中调用它。再次输入 `groovy Gwitter`，确认会出现 `File` 菜单，见图 4。选择 **File > Exit**，关闭这个应用程序。

图 **4. Gwitter** 的 **File** 菜单



再看看 清单 11。注意，`actionPerformed` 处理函数定义为闭包，而不是匿名类。与相应的 Java 代码相比，这样的代码更干净、更容易阅读。

现在，添加一些表单元素以执行搜索。

添加搜索面板

经验丰富的 Swing 开发人员善于用单独的 `JPanel` 组装出最终的应用程序。这些容器组件可以方便地把相似、相关的组件分组在一起。

例如，Gwitter 需要一个 `JTextField`（让用户能够输入搜索条件）和一个 `JButton`（用于提交请求）。把这两个组件分组在一个 `searchPanel` 闭包中是有意义的，见清单 12：

清单 **12.** 添加搜索面板

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  def searchField

  static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()

    def customMenuBar = {
      swingBuilder.menuBar{
        menu(text: "File", mnemonic: 'F') {
          menuItem(text: "Exit", mnemonic: 'X', actionPerformed: {d
        }
      }
    }

    def searchPanel = {
      swingBuilder.panel(constraints: BorderLayout.NORTH){
        searchField = textField(columns:15)
        button(text:"Search", actionPerformed:{ /* TODO */ } )
      }
    }

    swingBuilder.frame(title:"Gwitter",
                       defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                       size:[400,500],
                       show:true) {
      customMenuBar()
      searchPanel()
    }
  }
}
```

开始处理面板之后，就要选择适当的 `LayoutManger` 。在默认情况下， `JPanel` 使用 `FlowLayout` 。这意味着 `textField` 和 `button` 挨着水平排列。

`JFrame` 的 `contentPane` 不太一样 — 它在默认情况下使用 `BorderLayout` 。这意味着在框架中添加 `searchPanel` 时需要指定它应该出现在哪个区域： `NORTH` 、 `SOUTH` 、 `EAST` 、 `WEST` 或 `CENTER` 。（如果您的地理知识实在糟糕，也可以使用 `PAGE_START` 、 `PAGE_END` 、 `LINE_START` 、 `LINE_END` 和 `CENTER` ）。关于 Swing 中可用的各种 `LayoutManager` 的更多信息，请参见参考资料。

注意，`searchField` 变量是在类级声明的。因此，按钮等其他组件也可以访问它。其他组件都是匿名的。快速浏览一下类属性，就会看出某些组件比较重要。

您可能已经注意到按钮的 `actionPerformed` 监听器目前没有做任何事情。现在实际上还不需要它做什么。在实现它之前，需要在应用程序中添加另一个面板：用来显示搜索结果的面板。

添加结果面板

如清单 13 所示，像对待 `searchPanel` 那样，在嵌套的闭包中定义 `resultsPanel`。但是，这一次在这个面板中嵌套另一个容器：`JScrollPane`。这个组件可以根据需要显示和隐藏水平和垂直滚动条。`Search.byKeyword()` 方法调用的结果显示在名为 `resultsList` 的 `JList` 中。（`JList.setListData()` 方法接受一个 `Object[]` — 这就是 `Search.byKeyword()` 方法返回的结果）。

清单 **13.** 添加 **`resultsPanel`**

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

class Gwitter{
  def searchField
  def resultsList

  static void main(String[] args){
    def gwitter = new Gwitter()
    gwitter.show()
  }

  void show(){
    def swingBuilder = new SwingBuilder()

    def customMenuBar = {
      swingBuilder.menuBar{
        menu(text: "File", mnemonic: 'F') {
          menuItem(text: "Exit", mnemonic: 'X', actionPerformed: {
        }
      }
    }

    def searchPanel = {
      swingBuilder.panel(constraints: BorderLayout.NORTH){
        searchField = textField(columns:15)
        button(text:"Search", actionPerformed:{
          resultsList.listData = Search.byKeyword(searchField.text
      }
    }

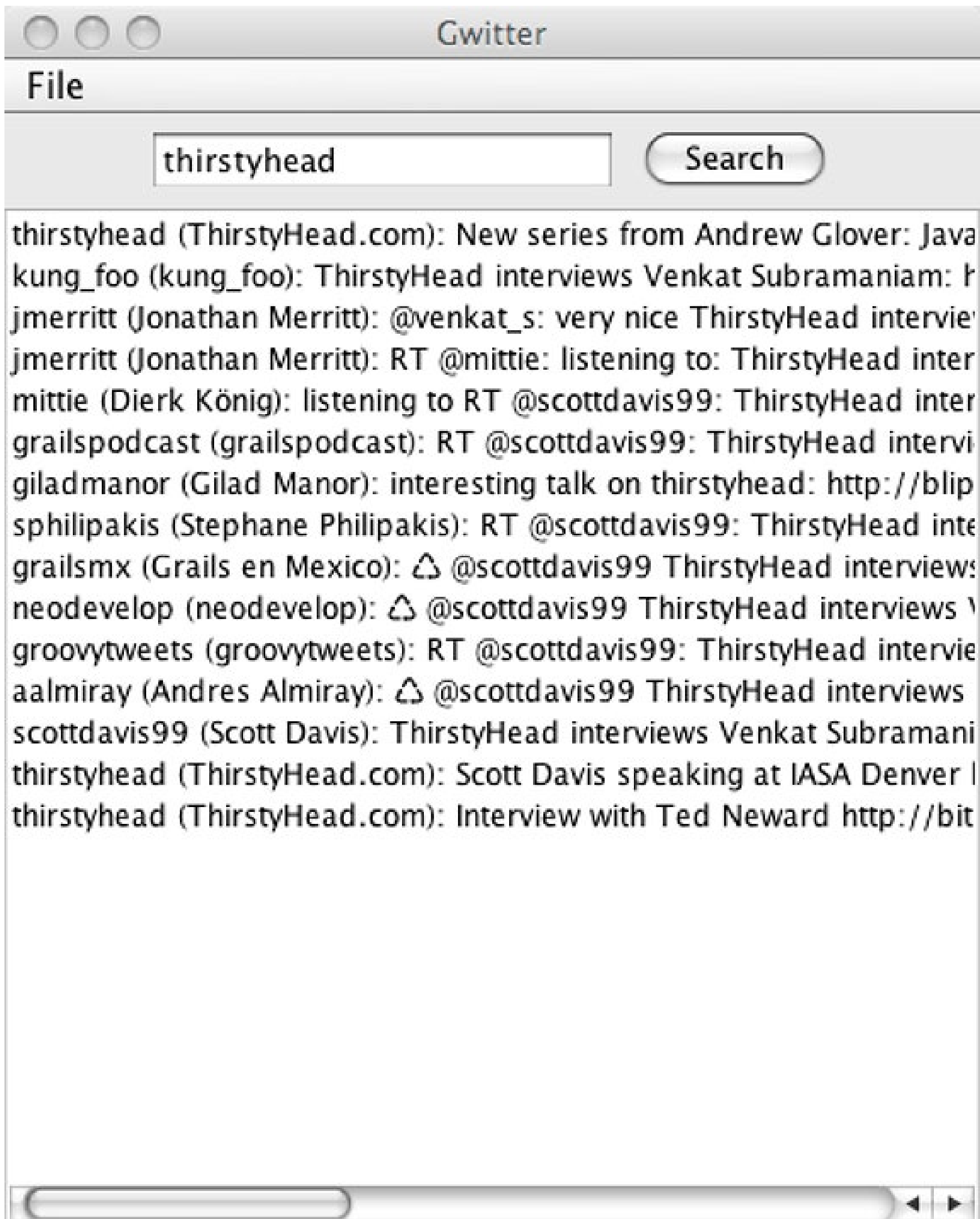
    def resultsPanel = {
      swingBuilder.scrollPane(constraints: BorderLayout.CENTER){
        resultsList = list()
      }
    }

    swingBuilder.frame(title:"Gwitter",
                       defaultCloseOperation:JFrame.EXIT_ON_CLOSE,
                       size:[400,500],
                       show:true) {
      customMenuBar()
      searchPanel()
      resultsPanel()
    }
  }
}
```

注意，与 `searchField` 一样， `resultsList` 变量是在类级定义的。 `searchPanel` 中按钮的 `actionPerformed` 处理函数使用这两个变量。

添加 `resultsPanel` 之后，Gwitter 现在有实际功能了。在命令提示上输入 `groovy Gwitter` ，检查它是否工作正常。搜索 *thirstyhead* 应该会产生图 5 所示的结果：

图 **5.** 搜索结果



现在可以宣布成功了，但是我想先解决两个问题。第一个问题是搜索按钮的 `actionPerformed` 处理函数可能会引起线程问题。另一个问题是这个应用程序太一般了。下面两节解决这些问题。

事件分派线程

Swing 的缺点在于，它期望图形设计师能够应付多线程问题，而这是应该由软件工程师处理的，或者期望软件工程师理解图形设计和易用性问题。

我不可能在短短几段文字中讨论 Swing 应用程序中的线程问题这么复杂的主题。只需指出基本的 Swing 应用程序本质上是单线程的。所有活动都在事件分派线程 (EDT) 上进行。当用户抱怨 Swing 应用程序反应迟缓或完全没有反应时，往往是因为某个开发新手在 EDT 上执行长时间的计算密集型的数据库查询或 Web 服务调用 — 这个线程也负责处理屏幕刷新、菜单单击等。我们无意中在搜索按钮的 `actionPerformed` 处理函数上犯了同样的错误。（您可以看出多么容易犯这种错误）。

好在 `javax.swing.SwingUtilities` 类提供了几个方便的方法 — `invokeAndWait()` 和 `invokeLater()` ，它们可以消除某些线程问题。可以使用这两个方法在 EDT 上同步或异步地执行操作。（关于 `SwingUtilities` 类的更多信息见 参考资料）。 `SwingBuilder` 让我们很容易调用这两个方法，还提供了第三个选择：可以简便地生成新线程以执行处理时间长的操作。

要想在 EDT 上执行同步调用 (`SwingUtilities.invokeAndWait()`)，可以把调用放在 `edt{}` 闭包中。要想在 EDT 上执行异步调用 (`SwingUtilities.invokeLater()`)，就把调用放在 `doLater{}` 闭包中。但是，我想让您体验一下第三个选择：生成新线程来处理 `Search.byKeyword()` 方法调用。为此，需要把代码放在 `doOutside{}` 闭包中，见清单 14：

清单 **14.** 使用 **`doOutside`** 闭包

```
def searchPanel = {
  swingBuilder.panel(constraints: BorderLayout.NORTH){
    searchField = textField(columns:15)
    button(text:"Search", actionPerformed:{
      doOutside{
        resultsList.listData = Search.byKeyword(searchField.text)
      }
    } )
  }
}
```

在像 Gwitter 这样简单的应用程序中，在 EDT 上执行 Web 服务调用很可能没什么不好的效果。但是，如果把这样的代码拿给 Swing 专家看，他们会用鄙视的目光看您，就像是您在快车道里慢慢地开车，或者把车停在商店停车场的残疾人专用车位上了。因为通过使用 `SwingBuilder` 很容易正确地处理线程，完全没有理由不这么做。

既然解决了线程问题，下面就让这个应用程序更漂亮一些。

给列表增加条纹效果

坦率地说，Gwitter 目前很难看。我要使用一些 HTML 代码做两个简单的改进，让外观和感觉好一些。 `JLabel` 可以显示基本的 HTML。按清单 15 调整 Tweet.groovy 的 `toString()` 方法。 `JList` 调用 `toString()` 方法显示结果。

清单 15. 在 `toString()` 方法中返回 HTML

```
class Tweet{
  String content
  String published
  String author

  String toString(){
    //return "${author}: ${content}"

    return """<html>
         <body>
           <p><b><i>${author}:</i></b></p>
           <p>${content}</p>
         </body>
       </html>"""
  }
}
```

下一个改进略微有点复杂。一种常用的 GUI 技巧是给长的列表或表格加上条纹效果。用不同的颜色显示奇数行和偶数行，这样读者更容易阅读。我在搜索引擎中搜索了 *JList stripes*，采纳了找到的第一篇文章中的建议。作者建议创建一个定制的 `DefaultListCellRenderer` 。我完全赞同他的意见并按原样借用他的示例代码（完整的文章见 参考资料）。

因为 Groovy 语法是 Java 语法的超集，所以可以把 Java 代码复制到 Groovy 文件中，不需要修改。如果有功能全面的构建系统，可以编译 Java 和 Groovy 代码，那么只需把这段代码留在 Java 文件中。但是，通过把代码文件的扩展名改为 .groovy，我可以运行所有未编译的 Gwitter 代码。我再次利用了 Java 语言和 Groovy 之间的无缝集成。可以在 Groovy 应用程序中不加修改地使用任何 Java 解决方案。

创建文件 StripeRenderer.groovy，添加清单 16 中的代码：

清单 16. 创建有条纹效果的 `CellRenderer`

```
import java.awt.*;
import javax.swing.*;

class StripeRenderer extends DefaultListCellRenderer {
    public Component getListCellRendererComponent(JList list, Objec
            int index, boolean isSelected, boolean cellHasFocus) {
        JLabel label = (JLabel) super.getListCellRendererComponent(
                index, isSelected, cellHasFocus);

        if(index%2 == 0) {
            label.setBackground(new Color(230,230,255));
        }

        label.setVerticalAlignment(SwingConstants.TOP);
        return label;
    }
}
```

有了 `StripeRenderer` 类之后，最后需要让 `JList` 使用它。按清单 17 调整 `resultsPanel`：

清单 17. 在 `JList` 中添加定制的 `CellRenderer`

```
def resultsPanel = {
  swingBuilder.scrollPane(constraints: BorderLayout.CENTER){
    //resultsList = list()
    resultsList =
       list(fixedCellWidth: 380, fixedCellHeight: 75, cellRenderer
  }
}
```

在命令提示上再次输入 `groovy Gwitter`。搜索 *thirstyhead* 应该会产生图 16 所示的结果：

图 6. 有条纹效果的结果



我可以花更多时间美化 Gwitter 的外观和感觉，但是我希望您对大约 50 行 Swing 代码（当然不包括支持类）所实现的效果印象深刻。

结束语

正如本文中指出的，Groovy 并不能降低 Swing 内在的复杂性，但是它可以显著降低语法复杂性。这让您能够留出时间应付更重要的问题。

如果本文引起了您对 Groovy 和 Swing 的兴趣，您应该好好研究一下 Griffon 项目（见 参考资料）。它提供许多优于 Grails 项目的功能和惯例，但是它基于 `SwingBuilder` 和 Groovy 而不是 Spring MVC 和 Hibernate。这个项目仍然处于

早期阶段（到编写本文时最新版本是 0.2），但是它已经很出色了，在 JavaOne 2009 上赢得了 Scripting Bowl for Groovy。另外，它提供的示例项目之一是 Greet，这是一个用 Groovy 实现的完整的 Twitter 客户机。

下一次，我将在 Gwitter 中添加一些必备特性：发布新 Tweet 的功能。在此过程中，您将学习如何处理基本的 HTTP 身份验证、执行 HTTP POST 以及使用与 `XmlSlurper` 相似的 `ConfigSlurper`。在此之前，我希望您探索应用 Groovy 的各种可能性。

下载

描述	名字	大小
文章示例的源代码	j-groovy09299.zip	24KB

实战 Groovy: @Delegate 注释

探索静态类型语言中的 *duck* 类型的极限

Scott Davis 将继续有关 Groovy 元编程的讨论，这一次他将深入研究 `@Delegate` 注释，@Delegate 注释模糊了数据类型和行为以及静态和动态类型之间的区别。

在过去几期 实战 *Groovy* 文章中，您已经了解了闭包和元编程之类的 Groovy 语言特性如何将动态功能添加到 Java™ 开发中。本文提供了更多这方面的内容。您将看到 `@Delegate` 注释如何演变自 `ExpandoMetaClass` 使用的 `delegate`。您将再一次领略到 Groovy 的动态功能如何使它成为单元测试的理想语言。

在 “使用闭包、ExpandoMetaClass 和类别进行元编程” 一文中，您了解了 `delegate` 的概念。当将一个 `shout()` 方法添加到 `java.lang.String` 的 `ExpandoMetaClass` 中时，您使用 `delegate` 来表示两个类之间的关系，如清单 1 所示：

清单 **1.** 使用 **`delegate`** 访问 **`String.toUpperCase()`**

```
String.metaClass.shout = {->
  return delegate.toUpperCase()
}

println "Hello MetaProgramming".shout()

//output
HELLO METAPROGRAMMING
```

您不能表示为 `this.toUpperCase()` ，因为 `ExpandoMetaClass` 并未包含 `toUpperCase()` 方法。类似地，也不能表示为 `super.toUpperCase()` ，因为 `ExpandoMetaClass` 没有扩展 `String` 。（事实上，它不可能扩展 `String` ，因为 `String` 是一个 `final` 类）。Java 语言并不具备用于表示这两个类之间的共生关系的词汇。这就是为什么 Groovy 要引入 `delegate` 概念。

关于本系列

Groovy 是在 Java 平台上运行的一种现代编程语言。它能够与现有 Java 代码无缝集成，同时引入了各种生动的新特性，比如闭包和元编程。简单来讲，Groovy 是 Java 语言的 21 世纪版本。

将任何新工具整合到开发工具包中的关键是知道何时使用它以及何时将它留在工具包中。Groovy 的功能可以非常强大，但唯一的条件是正确应用于适当的场景。因此，实战 *Groovy* 系列将探究 Groovy 的实际应用，以便帮助您了解何时以及如何成功使用它们。

在 Groovy 1.6 中， `@Delegate` 注释被添加到该语言中。（从 参考资料 部分可以获得添加到 Groovy 1.6 中的所有新注释的列表）。该注释允许您向任意 类添加一个或多个委托 — 而不仅仅是 `ExpandoMetaClass` 。

要充分地认识到 `@Delegate` 注释的威力，考虑 Java 编程中一个常见但复杂的任务：在 `final` 类的基础上创建一个新类。

复合模式和 `final` 类

假设您希望创建一个 `AllCapsString` 类，它具有 `java.lang.String` 的所有行为，唯一的不同是 — 正如名称暗示的那样 — 值始终以大写的形式返回。 `String` 是一个 `final` 类 — Java 演化到尽头的产物。清单 2 证明您无法直接扩展 `String` ：

清单 **2.** 扩展 **`final`** 类是不可能的

```
class AllCapsString extends String{
}

$ groovyc AllCapsString.groovy

org.codehaus.groovy.control.MultipleCompilationErrorsException:
startup failed, AllCapsString.groovy: 1: You are not allowed to
overwrite the final class 'java.lang.String'.
 @ line 1, column 1.
   class AllCapsString extends String{
   ^

1 error
```

这段代码无效，因此您的下一个最佳选择就是使用符合模式，如清单 3 所示（有关复合模式的更多信息，请参见 参考资料）：

清单 **3.** 对 **`String`** 类的新类型使用复合模式

```
class AllCapsString{
  final String body

  AllCapsString(String body){
    this.body = body.toUpperCase()
  }

  String toString(){
    body
  }

  //now implement all 72 String methods
  char charAt(int index){
    return body.charAt(index)
  }

  //snip...
  //one method down, 71 more to go...
}
```

因此，`AllCapsString` 类拥有 一个 `String` ，但是其行为 不同于 `String` ，除非您映射了所有 72 个 `String` 方法。要查看需要添加的方法，可以参考 Javadocs 中有关 `String` 的内容，或者运行清单 4 中的代码：

清单 4. 输出 **`String`** 类的所有方法

```
String.class.methods.eachWithIndex{method, i->
  println "${i} ${method}"
}

//output
0 public boolean java.lang.String.contentEquals(java.lang.CharSeque
1 public boolean java.lang.String.contentEquals(java.lang.StringBuf
2 public boolean java.lang.String.contains(java.lang.CharSequence)
...
```

将 72 个 `String` 方法手动添加到 `AllCapsString` 并不是一种明智的方法，而是在浪费开发人员的宝贵时间。这就是 `@Delegate` 注释发挥作用的时候了。

了解 `@Delegate`

`@Delegate` 是一个编译时注释，指导编译器将所有 delegate 的方法和接口推到外部类中。

在将 `@Delegate` 注释添加到 `body` 之前，编译 `AllCapsString` 并使用 `javap` 进行检验，看看大部分 `String` 方法是否缺失，如清单 5 所示：

清单 5. 在使用 `@Delegate` 前使用 `AllCapsString`

```
$ groovyc AllCapsString.groovy
$ javap AllCapsString
Compiled from "AllCapsString.groovy"
public class AllCapsString extends java.lang.Object
       implements groovy.lang.GroovyObject{
    public AllCapsString(java.lang.String);
    public java.lang.String toString();
    public final java.lang.String getBody();
    //snip...
```

现在，将 `@Delegate` 注释添加到 `body`，如清单 6 所示。重复 `groovyc` 和 `javap` 命令，将看到 `AllCapsString` 具有与 `java.lang.String` 相同的所有方法和接口。

清单 6. 使用 `@Delegate` 注释将 `String` 的所有方法推到周围的类中

```
class AllCapsString{
  @Delegate final String body

  AllCapsString(String body){
    this.body = body.toUpperCase()
  }

  String toString(){
    body
  }
}

$ groovyc AllCapsString.groovy
$ javap AllCapsString
Compiled from "AllCapsString.groovy"
public class AllCapsString extends java.lang.Object
       implements java.lang.CharSequence, java.lang.Comparable,
       java.io.Serializable,groovy.lang.GroovyObject{

    //NOTE: AllCapsString methods:
    public AllCapsString(java.lang.String);
    public java.lang.String toString();
    public final java.lang.String getBody();

    //NOTE: java.lang.String methods:
    public boolean contains(java.lang.CharSequence);
    public int compareTo(java.lang.Object);
    public java.lang.String toUpperCase();
    //snip...
```

然而，注意，您仍然可以调用 `getBody()` ，从而绕过被推入到环绕的 `AllCapsString` 类中的所有方法。通过将 `private` 添加到字段声明中 — `@Delegate final private String body` — 可以禁止显示普通的 getter/setter 方法。这将完成转换： `AllCapsString` 提供了 `String` 的全部行为，允许您根据情况覆盖 `String` 方法。

在静态语言中使用 duck 类型的限制

尽管 `AllCapsString` 目前拥有 `String` 的所有行为，但是它仍然不是一个真正的 `String` 。在 Java 代码中，无法使用 `AllCapsString` 作为 `String` 的临时替代，因为它并不是一个真正的 duck — 它只不过是冒充的。（动态语言被认为是使用 *duck* 类型；Java 语言使用静态 类型。参见 参考资料 获得更多与此有关的差异）。换句话说，由于 `AllCapsString` 并未真正扩展 `String` （或实现并不存在的 `Stringable` 接口），因此无法在 Java 代码中与 `String` 互相替换。清单 7 展示了在 Java 语言中将 `AllCapsString` 转换为 `String` 的失败例子：

清单 **7. Java** 语言中的静态类型阻止 **`AllCapsString`** 与 **`String`** 之间互相替换

```
public class JavaExample{
  public static void main(String[] args){
    String s = new AllCapsString("Hello");
  }
}

$ javac JavaExample.java
JavaExample.java:5: incompatible types
found   : AllCapsString
required: java.lang.String
    String s = new AllCapsString("Hello");
               ^
1 error
```

因此，通过允许您扩展被最初的开发人员明确禁止扩展的类，Groovy 的 `@Delegate` 并没有真正破坏 Java 的 `final` 关键字，但是您仍然可以获得与在不越界的情况下相同程度的威力。

请记住，您的类可以拥有多个 `delegate` 。假设您希望创建一个 `RemoteFile` 类，它将同时具有 `java.io.File` 和 `java.net.URL` 的特征。Java 语言并不支持多重继承，但是您可以非常接近一对 `@Delegate` ，如清单 8 所示。 `RemoteFile` 类不是 `File` 也不是 `URL` ，但是它却具有两者的行为。

清单 **8.** 多个 **`@Delegate`** 提供了多重继承的行为

```
class RemoteFile{
  @Delegate File file
  @Delegate URL url
}
```

如果 `@Delegate` 只能修改类的行为 — 而不是类型 — 这是否意味着对 Java 开发人员毫无价值？未必，即使是 Java 之类的静态类型语言也为 duck 类型提供了一种有限的形式，称为多态。

具有多态性的 duck

多态 — 该词源于希腊，用于描述“多种形状”— 意味着只要一组类通过实现相同接口显式地共享相同的行为，它们就可以互相替换着使用。换句话说，如果定义了一个 `Duck` 类型的变量（假设 `Duck` 是一个正式定义 `quack()` 和 `waddle()` 方法的接口），那么可以将 `new Mallard()` 、 `new GreenWingedTeal()` 或者（我最喜爱的） `new PekingWithHoisinSauce()` 分配给它。

通过将 delegate 类的方法和接口全部提升到其他类， `@Delegate` 注释为多态提供了完整的支持。这意味着如果 delegate 类实现了接口，您又回到了为它创建一个临时替代这件事上来。

`@Delegate` 和 `List` 接口

假设您希望创建一个名为 `FixedList` 的新类。它的行为应该类似 `java.util.ArrayList`，但是有一个重要的区别：您应当能够为可以添加到其中的元素的数量定义一个上限。这允许您创建一个 `sportsCar` 变量，该变量可以容纳两个乘客，但是不能比这再多了，`restaurantTable` 可以容纳 4 个用餐者，但是同样不能超过这个数字，以此类推。

`ArrayList` 类实现 `List` 接口。它为您提供了两个选项。您也可以让您的 `FixedList` 类实现 `List` 接口，但是您需要面对一项烦人的工作：为所有 `List` 方法提供一个实现。由于 `ArrayList` 并不是 `final` 类，另一个选择就是让 `FixedList` 扩展 `ArrayList`。这是一个非常有效的做法，但是如果（假设）`ArrayList` 被声明为 `final`，`@Delegate` 注释将提供第三个选择：通过将 `ArrayList` 作为 `FixedList` 的委托，您可以获得 `ArrayList` 的所有行为，同时自动实现 `List` 接口。

首先，使用一个 `ArrayList` 委托创建 `FixedList` 类，如清单 9 所示。`groovyc` / `javap` 是否可以检验 `FixedList` 不仅提供了与 `ArrayList` 相同的方法，还提供了相同的接口。

清单 **9.** 第一步创建 **`FixedList`** 类

```
class FixedList{
  @Delegate private List list = new ArrayList()
  final int sizeLimit

  /**
    * NOTE: This constructor limits the max size of the list,
    *  not just the initial capacity like an ArrayList.
    */
  FixedList(int sizeLimit){
    this.sizeLimit = sizeLimit
  }
}

$ groovyc FixedList.groovy
$ javap FixedList
Compiled from "FixedList.groovy"
public class FixedList extends java.lang.Object
             implements java.util.List,java.lang.Iterable,
             java.util.Collection,groovy.lang.GroovyObject{
    public FixedList(int);
    public java.lang.Object[] toArray(java.lang.Object[]);
    //snip..
```

目前我们还没有对 `FixedList` 的大小做任何限制，但这是一个很好的开始。如何确定 `FixedList` 的大小此时并不是固定的？您可以编写一些用后即扔的样例代码，但是如果 `FixedList` 将投入到生产中，您最好立即为其编写一些测试用例。

使用 `GroovyTestCase` 测试 `@Delegate`

要开始测试 `@Delegate`，编写一个单元测试，验证您可以将比您实际可添加的更多元素添加到 `FixedList`。清单 10 展示了这样一个测试：

清单 **10.** 首先编写一个失败的测试

```
class FixedListTest extends GroovyTestCase{

  void testAdd(){
    List threeStooges = new FixedList(3)
    threeStooges.add("Moe")
    threeStooges.add("Larry")
    threeStooges.add("Curly")
    threeStooges.add("Shemp")
    assertEquals threeStooges.sizeLimit, threeStooges.size()
  }
}

$ groovy FixedListTest.groovy

There was 1 failure:
1) testAdd(FixedListTest)junit.framework.AssertionFailedError:
   expected:<3> but was:<4>
```

似乎 `add()` 方法应当在 `FixedList` 中被重写，如清单 11 所示。重新运行这些测试仍然失败，但是这一次是因为抛出了异常。

清单 **11.** 重写 **`ArrayList`** 的 **`add()`** 方法

```
class FixedList{
  @Delegate private List list = new ArrayList()
  final int sizeLimit

  //snip...

  boolean add(Object element){
    if(list.size() < sizeLimit){
      return list.add(element)
    }else{
      throw new UnsupportedOperationException("Error adding ${eleme
                   " the size of this FixedList is limited to ${sizeL
    }
  }
}

$ groovy FixedListTest.groovy

There was 1 error:
1) testAdd(FixedListTest)java.lang.UnsupportedOperationException:
   Error adding Shemp: the size of this FixedList is limited to 3.
```

由于使用了 `GroovyTestCase` 的方便的 `shouldFail` 方法，您可以捕捉到这个预期的异常，如清单 12 所示，这一次您终于成功运行了测试：

清单 12. `shouldFail()` 方法捕捉到预期的异常

```
class FixedListTest extends GroovyTestCase{
  void testAdd(){
    List threeStooges = new FixedList(3)
    threeStooges.add("Moe")
    threeStooges.add("Larry")
    threeStooges.add("Curly")
    assertEquals threeStooges.sizeLimit, threeStooges.size()
    shouldFail(java.lang.UnsupportedOperationException){
      threeStooges.add("Shemp")
    }
  }
}
```

测试操作符重载

在 “美妙的操作符” 中，您了解到 Groovy 支持操作符重载。对于 `List` ，可以使用 `<<` 添加元素以及传统的 `add()` 方法。编写如清单 13 所示的快速单元测试，确定使用 `<<` 不会意外破坏 `FixedList` ：

清单 **13.** 测试操作员重载

```
class FixedListTest extends GroovyTestCase{

  void testOperatorOverloading(){
    List oneList = new FixedList(1)
    oneList << "one"
    shouldFail(java.lang.UnsupportedOperationException){
      oneList << "two"
    }
  }
}
```

这次测试的成功应该能够让您感到轻松一些。

您还可以测试出错的情况。比如，清单 14 测试了在创建包含一个负数元素的 `FixedList` 时出现的情况：

清单 **14.** 测试极端情况

```
class FixedListTest extends GroovyTestCase{
  void testNegativeSize(){
    List badList = new FixedList(-1)
    shouldFail(java.lang.UnsupportedOperationException){
      badList << "will this work?"
    }
  }
}
```

测试将一个元素插入到列表中间的情况

现在，您已经确信这个简单的重写过的 `add()` 方法可以正常工作，下一步是实现重载的 `add()` 方法，可以获取索引以及元素，如清单 15 所示：

清单 **15.** 使用索引添加元素

```
class FixedList{
  @Delegate private List list = new ArrayList()
  final int sizeLimit

  void add(int index, Object element){
    list.add(index, element)
    trimToSize()
  }

  private void trimToSize(){
    if(list.size() > sizeLimit){
      (sizeLimit..<list.size()).each{
        list.pop()
      }
    }
  }
}
```

注意，您可以（也应该）在任何可能的情况下使用 delegate 自带的功能 — 毕竟，这正是您优先选择 delegate 的原因。在这种情况下，您将让 `ArrayList` 执行添加操作，并去掉任何超出 `FixedList` 的大小的元素。（这个 `add()` 方法是否应该像另一个 `add()` 方法那样抛出一个 `UnsupportedOperationException`，您可以自己做出这个设计决策）。

`trimToSize()` 方法包含了一些值得关注的语法糖。首先，`pop()` 方法是由 Groovy 元编程到所有 `List` 中的内容。它删除了 `List` 中的最后一个元素，使用后进先出（last-in first-out，LIFO）的方式。

接下来，注意 `each` 循环中使用了一个 Groovy `range`。使用实数替换变量可能有助于使这一行为更加清晰。假设 `FixedList` 的 `sizeLimit` 的值为 `3`，并且在添加了新元素后，它的 `size()` 的值为 `5`。那么这个范围看上去应当类似于 `(3..5).each{}`。但是 `List` 使用的是基于 0 的标记法，因此列表中的元素不会拥有值为 `5` 的索引。通过指定 (3..<5).each{}，您将 5 排除到了这个范围之外。

编写两个测试，如清单 16 所示，检验新的重载后的 `add()` 方法是否如期望的那样运行：

清单 16. 测试将元素添加到 **`FixedList`** 中的情况

```
class FixedListTest extends GroovyTestCase{
  void testAddWithIndex(){
    List threeStooges = new FixedList(3)
    threeStooges.add("Moe")
    threeStooges.add("Larry")
    threeStooges.add("Curly")
    threeStooges.add(2,"Shemp")
    assertEquals 3, threeStooges.size()
    assertFalse threeStooges.contains("Curly")
  }

  void testAddWithIndexOnALessThanFullList(){
    List threeStooges = new FixedList(3)
    threeStooges.add("Curly")
    assertEquals 1, threeStooges.size()

    threeStooges.add(0, "Larry")
    assertEquals 2, threeStooges.size()
    assertEquals "Larry", threeStooges[0]

    threeStooges.add(0, "Moe")
    assertEquals 3, threeStooges.size()
    assertEquals "Moe", threeStooges[0]
    assertEquals "Larry", threeStooges[1]
    assertEquals "Curly", threeStooges[2]
  }
}
```

您是否注意到编写的测试代码的数量要多于生产代码？很好！我想说的是，对于每一段生产代码，您应当编写至少两倍数量的测试代码。

实现 `addAll()` 方法

要实现 `FixedList` 类，重写 `ArrayList` 中的 `addAll()` 方法，如清单 17 所示：

清单 17. 实现 `addAll()` 方法

```
class FixedList{
  @Delegate private List list = new ArrayList()
  final int sizeLimit

  boolean addAll(Collection collection){
    def returnValue = list.addAll(collection)
    trimToSize()
    return returnValue
  }

  boolean addAll(int index, Collection collection){
    def returnValue = list.addAll(index, collection)
    trimToSize()
    return returnValue
  }
}
```

现在编写相应的单元测试，如清单 18 所示：

清单 18. 测试 `addAll()` 方法

```
class FixedListTest extends GroovyTestCase{
  void testAddAll(){
    def quartet = ["John", "Paul", "George", "Ringo"]
    def trio = new FixedList(3)
    trio.addAll(quartet)
    assertEquals 3, trio.size()
    assertFalse trio.contains("Ringo")
  }

  void testAddAllWithIndex(){
    def quartet = new FixedList(4)
    quartet << "John"
    quartet << "Ringo"
    quartet.addAll(1, ["Paul", "George"])
    assertEquals "John", quartet[0]
    assertEquals "Paul", quartet[1]
    assertEquals "George", quartet[2]
    assertEquals "Ringo", quartet[3]
  }
}
```

您现在完成了全部工作。感谢 `@Delegate` 注释的强大威力，我们只使用大约 50 代码就创建了 `FixedList` 类。感谢 `GroovyTestCase` 使我们能够测试代码，从而允许您将其放入到生产环境中，并且确信它可以按照期望的那样操作。清单 19 展示了完整的 `FixedList` 类：

清单 19. 完整的 `FixedList` 类

```
class FixedList{
  @Delegate private List list = new ArrayList()
  final int sizeLimit

  /**
    * NOTE: This constructor limits the max size of the list,
    *  not just the initial capacity like an ArrayList.
    */
  FixedList(int sizeLimit){
    this.sizeLimit = sizeLimit
  }

  boolean add(Object element){
    if(list.size() < sizeLimit){
      return list.add(element)
    }else{
      throw new UnsupportedOperationException("Error adding ${eleme
        " the size of this FixedList is limited to ${sizeLimit}.")
    }
  }

  void add(int index, Object element){
    list.add(index, element)
    trimToSize()
  }

  private void trimToSize(){
    if(list.size() > sizeLimit){
      (sizeLimit..<list.size()).each{
        list.pop()
      }
    }
  }

  boolean addAll(Collection collection){
    def returnValue = list.addAll(collection)
    trimToSize()
    return returnValue
  }

  boolean addAll(int index, Collection collection){
    def returnValue = list.addAll(index, collection)
    trimToSize()
    return returnValue
  }

  String toString(){
    return "FixedList size: ${sizeLimit}\n" + "${list}"
  }
}
```

结束语

通过将新的行为 添加到类中而不是转换其类型，Groovy 的元编程功能实现了一组全新的动态可能性，同时不会违背 Java 语言的静态类型系统的规则。通过使用 `ExpandoMetaClass`（让您能够通过执行映射将任何新方法添加到现有类）和 `@Delegate`（让您能够通过外部包装类公开复合内部类的功能），Groovy 让 JVM 焕发新光彩。

在下一期文章中，我将演示一个得益于 Groovy 的灵活语法 Swing 而重新焕发生机的旧有技术。是的，Swing 的复杂性因为 Groovy 的 `SwingBuilder` 而消失。这使得桌面开发变得更加有趣和简单。到那时，希望您能够发现大量有关 Groovy 的实际应用。

下载

描述	名字	大小
本文示例的源代码	j-pg08259.zip	5KB

实战 **Groovy:** 使用闭包、**ExpandoMetaClass** 和类别进行元编程

随心所欲添加方法

进入到 Groovy 风格的元编程世界。在运行时向类动态添加方法的能力 — 甚至 Java™ 类以及 `final` Java 类 — 强大到令人难以置信。不管是用于生产代码、单元测试或介于两者之间的任何内容，即使是最缺乏热情的 Java 开发人员也会对 Groovy 的元编程能力产生兴趣。

人们一直以来都认为 Groovy 是一种面向 JVM 的动态 编程语言。在这期 实战 *Groovy* 文章中，您将了解元编程— Groovy 在运行时向类动态添加新方法的能力。它的灵活性远远超出了标准 Java 语言。通过一系列代码示例（都可以通过 下载 获得），将认识到元编程是 Groovy 的最强大、最实用的特性之一。

建模

程序员的工作就是使用软件建模真实的世界。对于真实世界中存在的简单域 — 比如具有鳞片或羽毛的动物通过产卵繁育后代，而具有毛皮的动物则通过产仔繁殖 — 可以很容易地使用软件对行为进行归纳，如清单 1 所示：

清单 **1.** 使用 **Groovy** 对动物进行建模

```
class ScalyOrFeatheryAnimal{
  ScalyOrFeatheryAnimal layEgg(){
    return new ScalyOrFeatheryAnimal()
  }
}

class FurryAnimal{
  FurryAnimal giveBirth(){
    return new FurryAnimal()
  }
}
```

关于本系列

Groovy 是在 Java 平台上运行的一种现代编程语言。它能够与现有 Java 代码无缝集成，同时引入了各种生动的新特性，比如闭包和元编程。简单来讲，Groovy 是 Java 语言的 21 世纪版本。

将任何新工具整合到开发工具包中的关键是知道何时使用它以及何时将它留在工具包中。Groovy 的功能可以非常强大，但惟一的条件是正确应用于适当的场景。因此，*实战 Groovy* 系列将探究 Groovy 的实际应用，以便帮助您了解何时以及如何成功使用它们。

不幸的是，真实的世界总是充满了例外和极端情况 — 鸭嘴兽既有皮毛，又通过产卵繁殖后代。我们精心考虑的每一项软件抽象几乎都存在与之相反的方面。

如果用来建模域的软件语言由于太过死板而无法处理不可避免的例外情况，那么最终的情形就像是受雇于一个小官僚机构的固执的公务员 —“对不起，Platypus 先生，如果要想我们的系统可以跟踪到您的话，您必须会生孩子。”

另一方面，Groovy 之类的动态语言为您提供了灵活性，使您能够更加准确地使用软件建模现实世界，而不是预先作出假设（并且通常是无效的），让现实向您妥协。如果 `Platypus` 类需要一个 `layEgg()` 方法，Groovy 可以满足要求，如清单 2 所示：

清单 **2.** 动态添加 **`layEgg()`** 方法

```
Platypus.metaClass.layEgg = {->
  return new FurryAnimal()
}

def baby = new Platypus().layEgg()
```

如果觉得这里举的有关动物的例子有些浅显，那么考虑 Java 语言中最常用的一个类：`String`。

Groovy 为 `java.lang.String` 提供的新方法

使用 Groovy 的乐趣之一就在于它添加到 `java.lang.String` 中的新方法。`padRight()` 和 `reverse()` 等方法提供了简单的 `String` 转换，如清单 3 所示。（有关 GDK 添加到 `String` 的所有新方法的列表的链接，见参考资料。正如 GDK 在其首页中所说，“本文档描述了添加到 JDK 并更具 groovy 特征的方法。”）

清单 **3. Groovy** 添加到 **`String`** 的方法

```
println "Introduction".padRight(15, ".")
println "Introduction".reverse()

//output
Introduction...
noitcudortnI
```

但是添加到 `String` 的方法并不仅限于简单的功能。如果 `String` 是一个组织良好的 URL，那么只需一行代码，您就可以将 `String` 转换为 `java.net.URL` 并返回 HTTP GET 请求的结果，如清单 4 所示：

清单 **4.** 发出 **HTTP GET** 请求

```
println "http://thirstyhead.com".toURL().text

//output
<html>
  <head>
    <title>ThirstyHead: Training done right.</title>
<!-- snip -->
```

再举一个例子，运行一个本地 shell 就像发出远程网络调用那么简单。一般情况下我将在命令提示中输入 `ifconfig en0` 以检查网卡的 TCP/IP 设置。（如果您使用的是 Windows® 而不是 Mac OS X 或 Linux®，那么尝试使用 `ipconfig` ）。在 Groovy 中，我可以通过编程的方式完成同样的事情，参见清单 5：

清单 **5.** 在 **Groovy** 中发出一个 **shell** 命令

```
println "ifconfig en0".execute().text

//output
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1
    ether 00:17:f2:cb:bc:6b
    media: autoselect status: inactive
  //snip
```

我并没有说 Groovy 的优点在于您不能 使用 Java 语言做同样的事情。您当然可以。Groovy 的优点在于这些方法似乎可以直接添加到 `String` 类 — 这绝非易事，因为 `String` 是 `final` 类。（稍后将详细讨论这点）。清单 6 展示了 Java 中的相应内容 `String.execute().text` ：

清单 **6.** 使用 **Java** 语言发出 **shell** 命令

```
Process p = new ProcessBuilder("ifconfig", "en0").start();
BufferedReader br = new BufferedReader(new InputStreamReader(p.getI
String line = br.readLine();
while(line != null){
  System.out.println(line);
  line = br.readLine();
}
```

这看上去有点像在机动车辆管理局的各个窗口之间辗转，不是吗？“对不起，先生，要查看您请求的 `String` ，首先需要去别处获得一个 `BufferedReader` 。”

是的，您可以构建方便的方法和实用类来帮助将这个问题抽象出来，但是惟一的 `com.mycompany.StringUtil` 替代方法就是使用一个类来代替将方法直接添加到所属位置的行为： `String` 类。（当然就是 `Platypus.layEgg()` ！）

那么 Groovy 究竟如何做 — 将新方法添加到无法扩展的类，或直接进行修改？要理解这一点，需要了解 *closures* 和 `ExpandoMetaClass` 。

闭包和 `ExpandoMetaClass`

Groovy 提供了一种无害的但功能强大的语言特性 — 闭包 — 如果没有它的话，鸭嘴兽将永远无法下蛋。简单来说，闭包就是指定的一段可执行代码。它是一个未包含在类中的方法。清单 7 演示了一个简单闭包：

清单 **7.** 一个简单闭包

```
def shout = {src->
  return src.toUpperCase()
}

println shout("Hello World")

//output
HELLO WORLD
```

拥有一个独立的方法当然很棒，但是与将方法放入到现有类的能力相比，还是有些逊色。考虑清单 8 中的代码，其中并未创建接受 `String` 作为参数的方法，相反，我将方法直接添加到 `String` 类：

清单 **8.** 将 **`shout`** 方法添加到 **`String`**

```
String.metaClass.shout = {->
  return delegate.toUpperCase()
}

println "Hello MetaProgramming".shout()

//output
HELLO METAPROGRAMMING
```

未包含任何参数的 `shout()` 闭包被添加到 `String` 的 `ExpandoMetaClass` (EMC) 中。每个类 — 包括 Java 和 Groovy — 都包含在一个 EMC 中，EMC 将拦截对它的方法调用。这意味着即使 `String` 为 `final`，仍然可以将方法添加到其 EMC 中。因此，现在看上去仿佛 `String` 有一个 `shout()` 方法。

由于 Java 语言中不存在这种关系，因此 Groovy 必须引入一个新的概念：委托（*delegate*）。`delegate` 是 EMC 所围绕的类。

首先了解到方法调用包含在 EMC 中，然后了解了 `delegate`，您就可以执行任何有趣的操作。比如，注意清单 9 然后重新定义 `String` 的 `toUpperCase()` 方法：

清单 **9.** 重新定义 **`toUpperCase()`** 方法

```
String.metaClass.shout = {->
  return delegate.toUpperCase()
}

String.metaClass.toUpperCase = {->
  return delegate.toLowerCase()
}

println "Hello MetaProgramming".shout()

//output
hello metaprogramming
```

这个操作看上去仍然有些不严谨（甚至有些危险！）。尽管现实中很少需要修改 `toUpperCase()` 方法的行为，但是想象一下为代码单元测试带来的好处？元编程提供了快速、简单的方法，使潜在的随机行为具有了必然性。比如，清单 10 演示了 `Math` 类的静态 `random()` 方法被重写：

清单 **10.** 重写 **`Math.random()`** 方法

```
println "Before metaprogramming"
3.times{
  println Math.random()
}

Math.metaClass.static.random = {->
  return 0.5
}

println "After metaprogramming"
3.times{
  println Math.random()
}

//output
Before metaprogramming
0.3452
0.9412
0.2932
After metaprogramming
0.5
0.5
0.5
```

现在，想像一下对发出开销较高的 SOAP 调用的类进行单元测试。无需创建接口和去掉整个模拟对象的存根 — 您可以有选择地重写方法并返回一个简单的模拟响应。（您将在下一小节看到使用 Groovy 实现单元测试和模拟的例子）。

Groovy 元编程是一种运行时行为 — 这个行为从程序启动一直持续到程序运行。但是如果希望对元编程进行更多的显示该怎么做（对于编写单元测试尤其重要）？在下一小节，您将了解揭秘元编程的秘密。

解密元编程

清单 11 封装了我在 `GroovyTestCase` 中编写的演示代码，这样就可以更加严格地对输出进行测试。（参见“实战 Groovy: 用 Groovy 更迅速地对 Java 代码进行单元测试”了解更多有关使用 `GroovyTestCase` 的信息）。

清单 **11.** 使用单元测试分析元编程

```
class MetaTest extends GroovyTestCase{

  void testExpandoMetaClass(){
    String message = "Hello"
    shouldFail(groovy.lang.MissingMethodException){
      message.shout()
    }

    String.metaClass.shout = {->
      delegate.toUpperCase()
    }

    assertEquals "HELLO", message.shout()

    String.metaClass = null
    shouldFail{
      message.shout()
    }
  }
}
```

在命令提示中输入 `groovy MetaTest` 以运行该测试。

注意，只需将 `String.metaClass` 设置为 `null`，就可以取消元编程。

但是，如果您不希望 `shout()` 方法出现在所有 `String` 中该怎么办呢？您可以仅调整单一实例的 EMC（而不是类），如清单 12 所示：

清单 **12.** 对单个实例进行元编程

```
void testInstance(){
  String message = "Hola"
  message.metaClass.shout = {->
    delegate.toUpperCase()
  }

  assertEquals "HOLA", message.shout()
  shouldFail{
    "Adios".shout()
  }
}
```

如果准备一次性添加或重写多个方法，清单 13 展示了如何以块的方式定义新方法：

清单 **13.** 一次性对多个方法进行元编程

```
void testFile(){
  File f = new File("nonexistent.file")
  f.metaClass{
    exists{-> true}
    getAbsolutePath{-> "/opt/some/dir/${delegate.name}"}
    isFile{-> true}
    getText{-> "This is the text of my file."}
  }

  assertTrue f.exists()
  assertTrue f.isFile()
  assertEquals "/opt/some/dir/nonexistent.file", f.absolutePath
  assertTrue f.text.startsWith("This is")
}
```

注意，我再也不关心文件是否存在于文件系统中。我可以将它发送给这个单元测试中的其他类，并且它表现得像一个眞正的文件。当 `f` 变量在测试结束时超出范围之后，还会执行定制行为。

尽管 `ExpandoMetaClass` 十分强大，但是 Groovy 提供了另一种元编程方法，使用了它独有的一组功能：类别（*category*）。

类别和 `use` 块

解释 `Category` 的最佳方法就是了解它的实际运行。清单 14 演示了使用 `Category` 来将 `shout()` 方法添加到 `String`：

清单 14. 使用一个 `Category` 进行元编程

```
class MetaTest extends GroovyTestCase{
  void testCategory(){
    String message = "Hello"
    use(StringHelper){
      assertEquals "HELLO", message.shout()
      assertEquals "GOODBYE", "goodbye".shout()
    }

    shouldFail{
      message.shout()
      "foo".shout()
    }
  }
}

class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}
```

如果曾经从事过 Objective-C 开发，那么应当对这个技巧感到熟悉。`StringHelper``Category` 是一个普通类 — 它不需要扩展特定的父类或实现特殊的接口。要向类型为 `T` 的特定类添加新方法，只需定义一个静态方法，它接受类型 `T` 作为第一个参数。由于 `shout()` 是一个接受 `String` 作为第一个参数的静态方法，因此所有封装到 `use` 块中的 `String` 都获得了一个 `shout()` 方法。

那么，什么时候应该选择 `Category` 而不是 EMC？EMC 允许您将方法添加到某个类的单一实例或所有实例中。可以看到，定义 `Category` 允许您将方法添加到特定 实例中 — 只限于 `use` 块内部的实例。

虽然 EMC 允许您动态定义新行为，然而 `Category` 允许您将行为保存到独立的类文件中。这意味着您可以在不同的情况下使用它：单元测试、生产代码，等等。定义单独类的开销在重用性方面获得了回报。

清单 15 演示了对同一个 `use` 块同时使用 `StringHelper` 和新创建的 `FileHelper`：

清单 15. 在 `use` 块中使用多个类别

```
class MetaTest extends GroovyTestCase{
  void testFileWithCategory(){
    File f = new File("iDoNotExist.txt")
    use(FileHelper, StringHelper){
      assertTrue f.exists()
      assertTrue f.isFile()
      assertEquals "/opt/some/dir/iDoNotExist.txt", f.absolutePath
      assertTrue f.text.startsWith("This is")

      assertTrue f.text.shout().startsWith("THIS IS")
    }

    assertFalse f.exists()
    shouldFail(java.io.FileNotFoundException){
      f.text
    }
  }
}

class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}

class FileHelper{
 static boolean exists(File f){
   return true
 }

 static String getAbsolutePath(File f){
   return "/opt/some/dir/${f.name}"
 }

 static boolean isFile(File f){
   return true
 }

 static String getText(File f){
   return "This is the text of my file."
 }
}
```

但是有关类别的最有趣的一点是它们的实现方式。EMC 需要使用闭包，这意味着您只能在 Groovy 中实现它们。由于类别仅仅是包含静态方法的类，因此可以用 Java 代码进行定义。事实上，可以在 Groovy 中重用现有的 Java 类 — 对元编程来说总是含义不明的类。

清单 16 演示了使用来自 Jakarta Commons Lang 包（见 参考资料）的类进行元编程。`org.apache.commons.lang.StringUtils` 中的所有方法都一致地遵守 `Category` 模式 — 静态方法接受 `String` 作为第一个参数。这意味着可以使用现成的 `StringUtils` 类作为 `Category`。

清单 **16.** 使用 **Java** 类进行元编程

```
import org.apache.commons.lang.StringUtils

class CommonsTest extends GroovyTestCase{
  void testStringUtils(){
    def word = "Introduction"

    word.metaClass.whisper = {->
      delegate.toLowerCase()
    }

    use(StringUtils, StringHelper){
      //from org.apache.commons.lang.StringUtils
      assertEquals "Intro...", word.abbreviate(8)

      //from the StringHelper Category
      assertEquals "INTRODUCTION", word.shout()

      //from the word.metaClass
      assertEquals "introduction", word.whisper()
    }
  }
}

class StringHelper{
  static String shout(String self){
    return self.toUpperCase()
  }
}
```

输入 `groovy -cp /jars/commons-lang-2.4.jar:. CommonsTest.groovy` 以运行测试（当然，您需要修改在系统中保存 JAR 的路径）。

元编程和 **REST**

为了不让您产生元编程只对 单元测试有用的误解，下面给出了最后一个例子。回忆一下 “实战 Groovy：构建和解析 XML” 中可以预报当天天气情况的 RESTful Yahoo! Web 服务。通过将上述文章中的 `XmlSlurper` 技巧与本文的元编程技巧结合起来，您就可以通过 10 行代码查看任何 ZIP 码所代表的位置的天气信息，如清单 17 所示：

清单 17. 添加一个 `weather` 方法

```
String.metaClass.weather={->
  if(!delegate.isInteger()){
    return "The weather() method only works with zip codes like '90
  }
  def addr = "http://weather.yahooapis.com/forecastrss?p=${delegate
  def rss = new XmlSlurper().parse(addr)
  def results = rss.channel.item.title
  results << "\n" + rss.channel.item.condition.@text
  results << "\nTemp: " + rss.channel.item.condition.@temp
}

println "80020".weather()

//output
Conditions for Broomfield, CO at 1:57 pm MDT
Mostly Cloudy
Temp: 72
```

可以看到，元编程提供了极好的灵活性。您可以使用本文介绍的任何（或所有）技巧来向任意数量的类添加方法。

结束语

要求世界因语言的限制而改变显然不切实际。使用软件建模真实世界意味着需要有足够灵活的工具来处理所有极端情况。幸运的是，使用 Groovy 提供的闭包、`ExpandoMetaClasses` 和类别，您就拥有了一组出色的工具，可以根据自己的意愿添加行为。

在下一期文章中，我将重新审视 Groovy 在单元测试方面的强大功能。使用 Groovy 编写测试会带来实际的效益，不管是 `GroovyTestCase` 或包含注释的 JUnit 4.x 测试用例。您将看到 GMock 的实际作用 — 一种使用 Groovy 编写的模拟框架。到那时，希望您能够发现 Groovy 的许多实际应用。

下载

描述	名字	大小
本文示例的源代码	j-pg06239.zip	7KB

实战 Groovy: 构建和解析 XML

简易 *XML* 操作

通过本文，您将了解使用 Groovy 分解 XML 是多么地容易。在本期的 实战 *Groovy* 中，作者 Scott Davis 演示了无论您是使用 `MarkupBuilder` 和 `StreamingMarkupBuilder` 创建 XML，还是使用 `XmlParser` 和 `XmlSlurper` 解析 XML，Groovy 都提供了一系列用于处理这类流行数据格式的工具。

XML 似乎已经由来已久。实际上，XML 在 2008 年迎来了它的 10 年庆典（参见 参考资料）。由于 Java™ 语言只比 XML 早几年出现，因此有人认为对于 Java 开发人员来说，XML 是 始终存在的。

关于本系列

Groovy 是在 Java 平台上运行的一种现代编程语言。它提供与已有 Java 代码的无缝集成，同时引入了各种生动的新特性，比如说闭包和元编程。简单来讲，Groovy 是 Java 语言的 21 世纪版本。

将任何新工具整合到开发工具包中的关键是知道何时使用它以及何时将它留在工具包中。Groovy 的功能可以非常强大，但惟一的条件是正确应用于适当的场景。因此，实战 *Groovy* 系列将探究 Groovy 的实际应用，以便帮助您了解何时以及如何成功使用它们。

Java 语言创始人 Sun Microsystems 一直是 XML 的积极支持者。毕竟，XML 的平台独立性承诺能与 Java 语言的 “编写一次，随处运行” 的口号完美契合。由于这两种技术具备一些相同的特性，您可能会认为 Java 语言和 XML 能很好地相处。事实上，在 Java 语言中解析和生成 XML 不但奇特而且还复杂。

幸运的是，Groovy 引入了一些全新的、更加合理的方法来创建和处理 XML。在一些示例的帮助下（均可通过 下载 获取），本文向您展示了如何通过 Groovy 简化 XML 的构建和解析。

比较 Java 和 Groovy XML 解析

在 “for each 剖析” 的结束部分，我提供了一个如清单 1 所示的简单 XML 文档。（这次，我添加了 `type` 属性，稍微增加了它的趣味性。）

清单 **1. XML** 文档，其中列出了我知道的语言

```
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
```

在 Java 语言中解析这个简单的 XML 文档却丝毫不简单，如清单 2 所示。它使用了 30 行代码来解析 5 行 XML 文件。

清单 **2.** 在 **Java** 中解析 **XML** 文件

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.IOException;

public class ParseXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse("src/languages.xml");

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    }catch(SAXException se) {
      se.printStackTrace();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

比较清单 2 中的 Java 代码和清单 3 中相应的 Groovy 代码：

清单 **3.** 在 **Groovy** 中解析 **XML**

```
def langs = new XmlParser().parse("languages.xml")
println "type = ${langs.attribute("type")}"
langs.language.each{
  println it.text()
}

//output:
type = current
Java
Groovy
JavaScript
```

Groovy 代码最出色的地方并不是它要比相应的 Java 代码简短很多 — 虽然使用 5 行 Groovy 代码解析 5 行 XML 是一个压倒性的优势。Groovy 代码最让我欣喜的一个地方就是它更具表达性。在编写 `langs.language.each` 时，我的感觉就像是在直接操作 XML。在 Java 版本中，您再也看不到 XML。

字符串变量和 XML

当您将 XML 存储在 `String` 变量而不是文件中时，在 Groovy 中使用 XML 的好处会变得更加明显。Groovy 的三重引号（在其他语言中通常称作 `HereDoc`）使得在内部存储 XML 变得非常轻松，如清单 4 所示。这与清单 3 中的 Groovy 示例之间的惟一区别就是将 `XmlParser` 方法调用从 `parse()`（它处理 `File`、`InputStreams`、`Reader` 和 `URI`）切换到 `parseText()`。

清单 4. 将 XML 存储在 Groovy 内部

```
def xml = """
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println "type = ${langs.attribute("type")}"
langs.language.each{
  println it.text()
}
```

注意，三重引号可以轻松地处理多行 XML 文档。`xml` 变量是一个真正的普通旧式（plain-old）`java.lang.String` — 您可以添加 `println xml.class` 自己进行验证。三重引号还可以处理 `type="current"` 的内部引号，而不会强制您像

在 Java 代码中那样使用反斜杠字符手动进行转义。

比较清单 4 中简洁的 Groovy 代码与清单 5 中相应的 Java 代码：

清单 5. 在 Java 代码内部存储 XML

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;

public class ParseXmlFromString {
  public static void main(String[] args) {
    String xml = "<langs type=\"current\">\n" +
            "  <language>Java</language>\n" +
            "  <language>Groovy</language>\n" +
            "  <language>JavaScript</language>\n" +
            "</langs>";

    byte[] xmlBytes = xml.getBytes();
    InputStream is = new ByteArrayInputStream(xmlBytes);

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse(is);

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    }catch(SAXException se) {
      se.printStackTrace();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

注意，`xml` 变量受到了针对内部引号和换行符的转义字符的污染。然而，更糟的是需要将 `String` 转换成一个 `byte` 数组，然后再转换成 `ByteArrayInputStream` 才能进行解析。`DocumentBuilder` 未提供将简单

`String` 作为 XML 解析的直观方法。

通过 `MarkupBuilder` 创建 XML

Groovy 相对 Java 语言最大的优势体现于在代码中创建 XML 文档。清单 6 显示了创建 5 行 XML 代码段所需的 50 行 Java 代码：

清单 6. 使用 Java 代码创建 XML

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

public class CreateXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.newDocument();

      Element langs = doc.createElement("langs");
      langs.setAttribute("type", "current");
      doc.appendChild(langs);

      Element language1 = doc.createElement("language");
      Text text1 = doc.createTextNode("Java");
      language1.appendChild(text1);
      langs.appendChild(language1);

      Element language2 = doc.createElement("language");
      Text text2 = doc.createTextNode("Groovy");
      language2.appendChild(text2);
      langs.appendChild(language2);

      Element language3 = doc.createElement("language");
      Text text3 = doc.createTextNode("JavaScript");
      language3.appendChild(text3);
      langs.appendChild(language3);

      // Output the XML
      TransformerFactory tf = TransformerFactory.newInstance();
      Transformer transformer = tf.newTransformer();
      transformer.setOutputProperty(OutputKeys.INDENT, "yes");
      StringWriter sw = new StringWriter();
      StreamResult sr = new StreamResult(sw);
      DOMSource source = new DOMSource(doc);
```

```
        transformer.transform(source, sr);
        String xmlString = sw.toString();
        System.out.println(xmlString);
      }catch(ParserConfigurationException pce) {
        pce.printStackTrace();
      } catch (TransformerConfigurationException e) {
        e.printStackTrace();
      } catch (TransformerException e) {
        e.printStackTrace();
      }
    }
  }
```

我知道一些人会立刻抱怨。许多第三方库都可以简化此代码 — JDOM 和 dom4j 是其中最流行的两个。但是，任何 Java 库都无法与使用 Groovy `MarkupBuilder` 的简洁性相比，如清单 7 所示：

清单 **7.** 使用 **Groovy** 创建 **XML**

```
def xml = new groovy.xml.MarkupBuilder()
xml.langs(type:"current"){
  language("Java")
  language("Groovy")
  language("JavaScript")
}
```

注意到这与 XML 代码的比率又重新回到了将近 1:1。更加重要的是，我可以再次查看 XML。当然，尖括号已经被替换为大括号，并且属性使用冒号（Groovy 的 `HashMap` 符号）而不是等号，但其基本结构在 Groovy 或 XML 中都是可以辨认的。它几乎类似于一个用于构建 XML 的 DSL，您认为呢？

Groovy 能够实现这种 `Builder` 魔法，因为它是一种动态的语言。另一方面，Java 语言则是静态的：Java 编译器将确保所有方法在您调用它们之前都是确实存在的。（如果您尝试调用不存在的方法，Java 代码甚至不进行编译，更不用说运行了。）但是，Groovy 的 `Builder` 证明，某种语言中的 bug 正是另一种语言的特性。如果您查阅 API 文档中的 `MarkupBuilder` 相关部分，您会发现它没有 `langs()` 方法、`language()` 方法或任何其他元素名称。幸运的是，Groovy 可以捕获这些不存在的方法调用，并采取一些有效的操作。对于 `MarkupBuilder` 的情况，它使用 phantom 方法调用并生成格式良好的 XML。

清单 8 对我刚才给出的简单的 `MarkupBuilder` 示例进行了扩展。如果您希望在 `String` 变量中捕获 XML 输出，则可以传递一个 `StringWriter` 到 `MarkupBuilder` 的构造函数中。如果您希望添加更多属性到 `langs` 中，只需要在传递时使用逗号将它们分开。注意，`language` 元素的主体是一个没有前置名称的值。您可以在相同的逗号分隔的列表中添加属性和主体。

清单 **8.** 经过扩展的 **`MarkupBuilder`** 示例

```
def sw = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(sw)
xml.langs(type:"current", count:3, mainstream:true){
  language(flavor:"static", version:"1.5", "Java")
  language(flavor:"dynamic", version:"1.6.0", "Groovy")
  language(flavor:"dynamic", version:"1.9", "JavaScript")
}
println sw

//output:
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
```

通过这些 `MarkupBuilder` 技巧，您可以实现一些有趣的功能。举例来说，您可以快速构建一个格式良好的 HTML 文档，并将它写出到文件中。清单 9 显示了相应的代码：

清单 9. 通过 `MarkupBuilder` 构建 HTML

```
def sw = new StringWriter()
def html = new groovy.xml.MarkupBuilder(sw)
html.html{
  head{
    title("Links")
  }
  body{
    h1("Here are my HTML bookmarks")
    table(border:1){
      tr{
        th("what")
        th("where")
      }
      tr{
        td("Groovy Articles")
        td{
          a(href:"http://ibm.com/developerworks", "DeveloperWorks")
        }
      }
    }
  }
}

def f = new File("index.html")
f.write(sw.toString())

//output:
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <h1>Here are my HTML bookmarks</h1>
    <table border='1'>
      <tr>
        <th>what</th>
        <th>where</th>
      </tr>
      <tr>
        <td>Groovy Articles</td>
        <td>
          <a href='http://ibm.com/developerworks'>DeveloperWorks</a
        </td>
      </tr>
    </table>
  </body>
</html>
```

图 1 显示了清单 9 所构建的 HTML 的浏览器视图：

图 **1.** 呈现的 **HTML**



使用 `StreamingMarkupBuilder` 创建 XML

`MarkupBuilder` 非常适合用于同步构建简单的 XML 文档。对于更加高级的 XML 创建，Groovy 提供了一个 `StreamingMarkupBuilder`。通过它，您可以添加各种各样的 XML 内容，比如说处理指令、名称空间和使用 `mkp` 帮助对象的未转义文本（非常适合 `CDATA` 块）。清单 10 展示了有趣的 `StreamingMarkupBuilder` 特性：

清单 **10.** 使用 `StreamingMarkupBuilder` 创建 **XML**

```
def comment = "<![CDATA[<!-- address is new to this release -->]]>'
def builder = new groovy.xml.StreamingMarkupBuilder()
builder.encoding = "UTF-8"
def person = {
  mkp.xmlDeclaration()
  mkp.pi("xml-stylesheet": "type='text/xsl' href='myfile.xslt'" )
  mkp.declareNamespace('':'http://myDefaultNamespace')
  mkp.declareNamespace('location':'http://someOtherNamespace')
  person(id:100){
    firstname("Jane")
    lastname("Doe")
    mkp.yieldUnescaped(comment)
    location.address("123 Main")
  }
}
def writer = new FileWriter("person.xml")
writer << builder.bind(person)

//output:
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type='text/xsl' href='myfile.xslt'?>
<person id='100'
        xmlns='http://myDefaultNamespace'
        xmlns:location='http://someOtherNamespace'>
  <firstname>Jane</firstname>
  <lastname>Doe</lastname>
  <![CDATA[<!-- address is new to this release -->]]>
  <location:address>123 Main</location:address>
</person>
```

注意，`StreamingMarkupBuilder` 直到您调用 `bind()` 方法时才会生成最终的 XML，该方法将接受标记和所有指令。这允许您异步构建 XML 文档的各个部分，并同时输出它们。（参见 参考资料 了解更多信息。）

理解 `XmlParser`

Groovy 为您提供了两种生成 XML — `MarkupBuilder` 和 `StreamingMarkupBuilder` 的方式 — 它们分别具备不同的功能。解析 XML 也同样如此。您可以使用 `XmlParser` 或者 `XmlSlurper` 。

`XmlParser` 提供了更加以程序员为中心的 XML 文档视图。如果您习惯于使用 `List` 和 `Map` （分别对应于 `Element` 和 `Attribute` ）来思考文档，则应该能够适应 `XmlParser` 。清单 11 稍微解析了 `XmlParser` 的结构：

清单 11. `XmlParser` 详细视图

```
def xml = """
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.getClass()
// class groovy.util.Node

println langs
/*
langs[attributes={type=current, count=3, mainstream=true};
      value=[language[attributes={flavor=static, version=1.5};
                      value=[Java]],
             language[attributes={flavor=dynamic, version=1.6.0};
                      value=[Groovy]],
             language[attributes={flavor=dynamic, version=1.9};
                      value=[JavaScript]]
            ]
]
*/
```

注意，`XmlParser.parseText()` 方法返回了一个 `groovy.util.Node` — 在本例中是 XML 文档的根 `Node` 。当您调用 `println langs` 时，它会调用 `Node.toString()` 方法，以便返回调试输出。要获取真实数据，您需要调用 `Node.attribute()` 或者 `Node.text()` 。

使用 `XmlParser` 获取属性

如前所述，您可以通过调用 `Node.attribute("key")` 来获取单独的属性。如果您调用 `Node.attributes()` ，它会返回包含所有 `Node` 的属性的 `HashMap` 。使用您在 “for each 剖析” 一文中所掌握的 `each` 闭包，遍历每个属性简直就是小菜一碟。清单 12 显示了一个相应的例子。（参见 参考资料，获取关于 `groovy.util.Node` 的 API 文档。）

清单 12. `XmlParser` 将属性作为 HashMap 对待

```
def langs = new XmlParser().parseText(xml)

println langs.attribute("count")
// 3

langs.attributes().each{k,v->
  println "-" * 15
  println k
  println v
}

//output:
---------------
type
current
---------------
count
3
---------------
mainstream
true
```

与操作属性相类似，`XmlParser` 为处理元素提供了更好的支持。

使用 `XmlParser` 获取元素

`XmlParser` 提供了一种直观的查询元素的方法，称作 `GPath` 。（它与 `XPath` 类似，仅在 Groovy 中得到了实现。）举例来说，清单 13 演示了我之前使用的 `langs.language` 结构返回了包含查询结构的 `groovy.util.NodeList` 。 `NodeList` 扩展了 `java.util.ArrayList` ，因此它基本上就是一个赋予了 `GPath` 超级权限的 `List` 。

清单 13. 使用 `GPath` 和 `XmlParser` 进行查询

```
def langs = new XmlParser().parseText(xml)

// shortcut query syntax
// on an anonymous NodeList
langs.language.each{
  println it.text()
}

// separating the query
// and the each closure
// into distinct parts
def list = langs.language
list.each{
  println it.text()
}

println list.getClass()
// groovy.util.NodeList
```

当然，`GPath` 是对 `MarkupBuilder` 的补充。它所采用的技巧与调用不存在的 phantom 方法相同，区别仅在于它用于查询已有的 XML 而不是动态地生成 XML。

知道 `GPath` 查询的结果是 `List` 之后，您可以让您的代码更加简练。Groovy 提供了一个 spread-dot 运算符。在单行代码中，它基本上能迭代整个列表并对每个项执行方法调用。结果将作为 `List` 返回。举例来说，如果您只关心对查询结果中的各个项调用 `Node.text()` 方法，那么清单 14 展示了如何在一行代码中实现它：

清单 **14.** 结合 **spread-dot** 运算符与 **`GPath`**

```
// the long way of gathering the results
def results = []
langs.language.each{
  results << it.text()
}

// the short way using the spread-dot operator
def values = langs.language*.text()
// [Java, Groovy, JavaScript]

// quickly gathering up all of the version attributes
def versions = langs.language*.attribute("version")
// [1.5, 1.6.0, 1.9]
```

和功能强大的 `XmlParser` 一样，`XmlSlurper` 也实现了更高级别的处理。

使用 `XmlSlurper` 解析 XML

在 清单 2 中，我说过 Groovy 给我的感觉是在直接操作 XML。 `XmlParser` 的功能相当不错，但它只允许您以编程的方式来操作 XML。您可以使用由 `Node` 组成的 `List` 以及由 `Attribute` 组成的 `HashMap` ，并且仍然需要调用 `Node.attribute()` 和 `Node.text()` 等方法才能获取核心数据。 `XmlSlurper` 将删除方法调用的最后痕迹，让您感觉就像是在直接处理 XML。

从技术上说， `XmlParser` 返回 `Node` 和 `NodeList` ，而 `XmlSlurper` 返回一个 `groovy.util.slurpersupport.GPathResult` 。但既然您已经知道，因此我希望您能忘记之前提到的 `XmlSlurper` 的实现细节。如果您能忽略其内部原理，那么将更好地领略其魔力。

清单 15 同时展示了一个 `XmlParser` 和一个 `XmlSlurper` ：

清单 15. `XmlParser` 和 `XmlSlurper`

```
def xml = """
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.attribute("count")
langs.language.each{
  println it.text()
}

langs = new XmlSlurper().parseText(xml)
println langs._cnnew1@count
langs.language.each{
  println it
}
```

注意， `XmlSlurper` 忽略了任何方法调用的概念。您并没有调用 `langs.attribute("count")` ，而是调用了 `langs.@count` 。 `@` 符号是从 `XPath` 借过来的，但其结果是，您感觉像是在直接操作属性（与调用 `attribute()` 方法相反）。您没有调用 `it.text()` ，而仅仅调用了 `it` 。我们假设您希望直接操作元素的内容。

现实中的 `XmlSlurper`

除了 `langs` 和 `language` 之外，这里还提供了实际的 `XmlSlurper` 示例。Yahoo! 以 RSS 提要的方式按 ZIP 码提供天气情况信息。当然，RSS 是 XML 中的一种专门术语。在 Web 浏览器中键入 `http://weather.yahooapis.com/forecastrss?p=80020` 。可以随意将 Broomfield, Colorado 的 ZIP 码换成您自己的。清单 16 显示了最终 RSS 提要的简单版本：

清单 **16.** 显示最新天气情况的 **Yahoo! RSS** 提要

```
<rss version="2.0"
     xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
     xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo! Weather - Broomfield, CO</title>
    <yweather:location city="Broomfield" region="CO"   country="US'
    <yweather:astronomy sunrise="6:36 am"   sunset="5:50 pm"/>

    <item>
      <title>Conditions for Broomfield, CO at 7:47 am MST</title>
      <pubDate>Fri, 27 Feb 2009 7:47 am MST</pubDate>
      <yweather:condition text="Partly Cloudy"
                          code="30"  temp="25"
                          date="Fri, 27 Feb 2009 7:47 am MST" />
    </item>
  </channel>
</rss>
```

您要做的第一件事就是通过编程来使用这个 RSS。创建一个名称为 weather.groovy 的文件，并添加如清单 17 所示的代码：

清单 **17.** 以编程的方式获取 **RSS**

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text
  println xml
}else{
  println "USAGE: weather zipcode"
}
```

在命令行中键入 `groovy weather 80020` ，确定您可以看到原始 RSS。

此脚本最重要的部分是 `url.toURL().text` 。 `url` 变量是一个格式良好的 `String` 。Groovy 在所有 `String` 中都添加了一个 `toURL()` 方法，用于将它们转换成 `java.net.URL` 。然后，Groovy 在所有 `URL` 中又添加了一个 `getText()` 方法，用于执行 `HTTP GET` 请求并将结构作为 `String` 返回。

现在，您已经将 RSS 存储在了 `xml` 变量中，并通过 `XmlSlurper` 实现了一些有趣的功能，如清单 18 所示：

清单 **18.** 使用 **`XmlSlurper`** 解析 **RSS**

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text

  def rss = new XmlSlurper().parseText(xml)
  println rss.channel.title
  println "Sunrise: ${rss.channel.astronomy.@sunrise}"
  println "Sunset: ${rss.channel.astronomy.@sunset}"
  println "Currently:"
  println "\t" + rss.channel.item.condition.@date
  println "\t" + rss.channel.item.condition.@temp
  println "\t" + rss.channel.item.condition.@text
}else{
  println "USAGE: weather zipcode"
}

//output:
Yahoo! Weather - Broomfield, CO
Sunrise: 6:36 am
Sunset: 5:50 pm
Currently:
   Fri, 27 Feb 2009 7:47 am MST
   25
   Partly Cloudy
```

`XmlSlurper` 让您可以自然地处理 XML，不是吗？您通过直接引用 <title> 元素来打印它 — `rss.channel.title` 。您使用一个简单的 `rss.channel.item.condition.@temp` 来去除 `temp` 属性。这与编程的感觉不同。它更像是在直接操作 XML。

您是否注意到 `XmlSlurper` 甚至忽略了名称空间？您在构造函数中启用名称空间感知，但我很少这样做。非常简单， `XmlSlurper` 能像切黄油那样分解 XML。

结束语

要在如今成为一名成功的开发人员，您需要一系列能简化 XML 处理的工具。Groovy 的 `MarkupBuilder` 和 `StreamingMarkupBuilder` 可以非常轻松地动态创建 XML。`XmlParser` 能为您提供由 `Element` 组成的 `List` 以及由 `Attribute` 组成的 `HashMap`，并且 `XmlSlurper` 可以让代码全部消失，让您感觉是在直接操作 XML。

如果没有 Groovy 的动态功能，XML 处理的强大功能将不可能实现。在下一章文章中，我将更加深入地探索 Groovy 的动态特性。您将了解元编程在 Groovy 中的工作原理，从标准 JDK 类（如 `String.toURL()` 和 `List.each()`）中添加的出色方法到您自己添加的自定义方法。在阅读了这两篇文章之后，我希望您能充分了解 Groovy 的 实际应用。

下载

描述	名字	大小
本文示例的源代码	j-pg05199.zip	6KB

实战 Groovy: for each 剖析

使用最熟悉的方法进行迭代

在这一期的 实战 *Groovy* 中，Scott Davis 提出了一组非常好的遍历方法，这些方法可以遍历数组、列表、文件、URL 以及很多其它内容。最令人印象深刻的是，Groovy 提供了一种一致的机制来遍历所有这些集合和其它内容。

迭代是编程的基础。您经常会遇到需要进行逐项遍历的内容，比如 `List` 、 `File` 和 JDBC `ResultSet` 。Java 语言几乎总是提供了某种方法帮助您逐项遍历所需的内容，但令人沮丧的是，它并没有给出一种标准方法。Groovy 的迭代方法非常实用，在这一点上，Groovy 编程与 Java 编程截然不同。通过一些代码示例（可从 下载 小节获得），本文将介绍 Groovy 的万能的 `each()` 方法，从而将 Java 语言的那些迭代怪癖抛在脑后。

Java 迭代策略

假设您有一个 Java 编程语言的 `java.util.List` 。清单 1 展示了在 Java 语言中如何使用编程实现迭代：

清单 **1. Java** 列表迭代

```
import java.util.*;

public class ListTest{
  public static void main(String[] args){
    List<String> list = new ArrayList<String>();
    list.add("Java");
    list.add("Groovy");
    list.add("JavaScript");

    for(Iterator<String> i = list.iterator(); i.hasNext();){
      String language = i.next();
      System.out.println("I know " + language);
    }
  }
}
```

由于提供了大部分集合类都可以共享的 `java.lang.Iterable` 接口，您可以使用相同的方法遍历 `java.util.Set` 或 `java.util.Queue` 。

关于本系列

Groovy 是一款运行在 Java 平台之上的现代编程语言。它能够与现有 Java 代码无缝集成，同时引入了闭包和元编程等出色的新特性。简而言之，Groovy 类似于 21 世纪的 Java 语言。

如果要将新工具集成到开发工具箱中，最关键的是理解什么时候需要使用它以及什么时候不适合使用它。Groovy 可以变得非常强大，但前提是它被适当地应用到合适的场景中。因此，*实战 Groovy* 系列旨在展示 Groovy 的实际使用，以及何时和如何成功应用它。

现在，假设该语言存储在 `java.util.Map` 中。在编译时，尝试对 `Map` 获取 `Iterator` 会导致失败 — `Map` 并没有实现 `Iterable` 接口。幸运的是，可以调用 `map.keySet()` 返回一个 `Set`，然后就可以继续处理。这些小差异可能会影响您的速度，但不会妨碍您的前进。需要注意的是，`List`、`Set` 和 `Queue` 实现了 `Iterable`，但是 `Map` 没有 — 即使它们位于相同的 `java.util` 包中。

现在假设该语言存在于 `String` 数组中。数组是一种数据结构，而不是类。不能对 `String` 数组调用 `.iterator()`，因此必须使用稍微不同的迭代策略。您再一次受到阻碍，但可以使用如清单 2 所示的方法解决问题：

清单 **2. Java** 数组迭代

```
public class ArrayTest{
  public static void main(String[] args){
    String[] list = {"Java", "Groovy", "JavaScript"};

    for(int i = 0; i < list.length; i++){
      String language = list[i];
      System.out.println("I know " + language);
    }
  }
}
```

但是等一下 — 使用 Java 5 引入的 for-each 语法怎么样（参见 参考资料）？它可以处理任何实现 `Iterable` 的类和数组，如清单 3 所示：

清单 **3. Java** 语言的 **for-each** 迭代

```
import java.util.*;

public class MixedTest{
  public static void main(String[] args){
    List<String> list = new ArrayList<String>();
    list.add("Java");
    list.add("Groovy");
    list.add("JavaScript");

    for(String language: list){
      System.out.println("I know " + language);
    }

    String[] list2 = {"Java", "Groovy", "JavaScript"};
    for(String language: list2){
      System.out.println("I know " + language);
    }
  }
}
```

因此，您可以使用相同的方法遍历数组和集合（ `Map` 除外）。但是如果语言存储在 `java.io.File` ，那该怎么办？如果存储在 JDBC `ResultSet` ，或者存储在 XML 文档、 `java.util.StringTokenizer` 中呢？面对每一种情况，必须使用一种稍有不同的迭代策略。这样做并不是有什么特殊目的 — 而是因为不同的 API 是由不同的开发人员在不同的时期开发的 — 但事实是，您必须了解 6 个 Java 迭代策略，特别是使用这些策略的特殊情况。

Eric S. Raymond 在他的 *The Art of Unix Programming*（参见 参考资料）一书中解释了 “最少意外原则”。他写道，“要设计可用的接口，最好不要设计全新的接口模型。新鲜的东西总是难以入门；会为用户带来学习的负担，因此应当尽量减少新内容。”Groovy 对迭代的态度正是采纳了 Raymond 的观点。在 Groovy 中遍历几乎任何结构时，您只需要使用 `each()` 这一种方法。

Groovy 中的列表迭代

首先，我将 清单 3 中的 `List` 重构为 Groovy。在这里，只需要直接对列表调用 `each()` 方法并传递一个闭包，而不是将 `List` 转换成 `for` 循环（顺便提一句，这样做并不是特别具有面向对象的特征，不是吗）。

创建一个名为 listTest.groovy 的文件并添加清单 4 中的代码：

清单 **4. Groovy** 列表迭代

```
def list = ["Java", "Groovy", "JavaScript"]
list.each{language->
  println language
}
```

清单 4 中的第一行是 Groovy 用于构建 `java.util.ArrayList` 的便捷语法。可以将 `println list.class` 添加到此脚本来验证这一点。接下来，只需对列表调用 `each()`，并在闭包体内输出 `language` 变量。在闭包的开始处使用 `language->` 语句命名 `language` 变量。如果没有提供变量名，Groovy 提供了一个默认名称 `it`。在命令行提示符中输入 `groovy listTest` 运行 listTest.groovy。

清单 5 是经过简化的 清单 4 代码版本：

清单 **5.** 使用 **Groovy** 的 `it` 变量的迭代

```
// shorter, using the default it variable
def list = ["Java", "Groovy", "JavaScript"]
list.each{ println it }

// shorter still, using an anonymous list
["Java", "Groovy", "JavaScript"].each{ println it }
```

Groovy 允许您对数组和 `List` 交替使用 `each()` 方法。为了将 `ArrayList` 改为 `String` 数组，必须将 `as String[]` 添加到行末，如清单 6 所示：

清单 **6. Groovy** 数组迭代

```
def list = ["Java", "Groovy", "JavaScript"] as String[]
list.each{println it}
```

在 Groovy 中普遍使用 `each()` 方法，并且 getter 语法非常便捷（`getClass()` 和 `class` 是相同的调用），这使您能够编写既简洁又富有表达性的代码。例如，假设您希望利用反射显示给定类的所有公共方法。清单 7 展示了这个例子：

清单 **7. Groovy** 反射

```
def s = "Hello World"
println s
println s.class
s.class.methods.each{println it}

//output:
$ groovy reflectionTest.groovy
Hello World
class java.lang.String
public int java.lang.String.hashCode()
public volatile int java.lang.String.compareTo(java.lang.Object)
public int java.lang.String.compareTo(java.lang.String)
public boolean java.lang.String.equals(java.lang.Object)
...
```

脚本的最后一行调用 `getClass()` 方法。`java.lang.Class` 提供了一个 `getMethods()` 方法，后者返回一个数组。通过将这些操作串连起来并对 `Method` 的结果数组调用 `each()`，您只使用了一行代码就完成了大量工作。

但是，与 Java for-each 语句不同的是，万能的 `each()` 方法并不仅限于 `List` 和数组。在 Java 语言中，故事到此结束。然而，在 Groovy 中，故事才刚刚开始。

Map 迭代

从前文可以看到，在 Java 语言中，无法直接迭代 `Map`。在 Groovy 中，这完全不是问题，如清单 8 所示：

清单 **8. Groovy map** 迭代

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
map.each{ println it }
```

要处理名称/值对，可以使用隐式的 `getKey()` 和 `getValue()` 方法，或在包的开头部分显式地命名变量，如清单 9 所示：

清单 **9.** 从 **map** 获得键和值

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
map.each{
  println it.key
  println it.value
}

map.each{k,v->
  println k
  println v
}
```

可以看到，迭代 `Map` 和迭代其它任何集合一样自然。

在继续研究下一个迭代例子前，应当了解 Groovy 中有关 `Map` 的另一个语法。与在 Java 语言中调用 `map.get("Java")` 不一样，可以简化对 `map.Java` 的调用，如清单 10 所示：

清单 10. 获得 map 值

```
def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]

//identical results
println map.get("Java")
println map.Java
```

不可否认，Groovy 针对 `Map` 的这种便捷语法非常酷，但这也是在对 `Map` 使用反射时引起一些常见问题的原因。对 `list.class` 的调用将生成 `java.util.ArrayList` ，而调用 `map.class` 返回 `null` 。这是因为获得 map 元素的便捷方法覆盖了实际的 getter 调用。 `Map` 中的元素都不具有 `class` 键，因此调用实际会返回 `null` ，如清单 11 的示例所示：

清单 11. Groovy map 和 `null`

```
def list = ["Java", "Groovy", "JavaScript"]
println list.class
// java.util.ArrayList

def map = ["Java":"server", "Groovy":"server", "JavaScript":"web"]
println map.class
// null

map.class = "I am a map element"
println map.class
// I am a map element

println map.getClass()
// class java.util.LinkedHashMap
```

这是 Groovy 比较罕见的打破“最少意外原则”的情况，但是由于从 map 获取元素要比使用反射更加常见，因此我可以接受这一例外。

String 迭代

现在您已经熟悉 `each()` 方法了，它可以出现在所有相关的位置。假设您希望迭代一个 `String` ，并且是逐一迭代字符，那么马上可以使用 `each()` 方法。如清单 12 所示：

清单 **12.** **`String`** 迭代

```
def name = "Jane Smith"
name.each{letter->
  println letter
}
```

这提供了所有的可能性，比如使用下划线替代所有空格，如清单 13 所示：

清单 **13.** 使用下划线替代空格

```
def name = "Jane Smith"
println "replace spaces"
name.each{
  if(it == " "){
    print "_"
  }else{
    print it
  }
}

// output
Jane_Smith
```

当然，在替换一个单个字母时，Groovy 提供了一个更加简洁的替换方法。您可以将清单 13 中的所有代码合并为一行代码：`"Jane Smith".replace(" ", "_")`。但是对于更复杂的 `String` 操作，`each()` 方法是最佳选择。

Range 迭代

Groovy 提供了原生的 `Range` 类型，可以直接迭代。使用两个点分隔的所有内容（比如 `1..10`）都是一个 `Range`。清单 14 展示了这个例子：

清单 **14. Range** 迭代

```
def range = 5..10
range.each{
  println it
}

//output:
5
6
7
8
9
10
```

`Range` 不局限于简单的 `Integer`。考虑清单 15 在的代码，其中迭代 `Date` 的 `Range`：

清单 **15.** **`Date`** 迭代

```
def today = new Date()
def nextWeek = today + 7
(today..nextWeek).each{
  println it
}

//output:
Thu Mar 12 04:49:35 MDT 2009
Fri Mar 13 04:49:35 MDT 2009
Sat Mar 14 04:49:35 MDT 2009
Sun Mar 15 04:49:35 MDT 2009
Mon Mar 16 04:49:35 MDT 2009
Tue Mar 17 04:49:35 MDT 2009
Wed Mar 18 04:49:35 MDT 2009
Thu Mar 19 04:49:35 MDT 2009
```

可以看到，`each()` 准确地出现在您所期望的位置。Java 语言缺乏原生的 `Range` 类型，但是提供了一个类似地概念，采取 `enum` 的形式。毫不奇怪，在这里 `each()` 仍然派得上用场。

Enumeration 类型

Java `enum` 是按照特定顺序保存的随意的值集合。清单 16 展示了 `each()` 方法如何自然地配合 `enum`，就好象它在处理 `Range` 操作符一样：

清单 **16.** **`enum`** 迭代

```
enum DAY{
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}

DAY.each{
  println it
}

(DAY.MONDAY..DAY.FRIDAY).each{
  println it
}
```

在 Groovy 中，有些情况下，`each()` 这个名称远未能表达它的强大功能。在下面的例子中，将看到使用特定于所用上下文的方法对 `each()` 方法进行修饰。Groovy `eachRow()` 方法就是一个很好的例子。

SQL 迭代

在处理关系数据库表时，经常会说“我需要针对表中的每一行执行操作”。比较一下前面的例子。您很可能会说“我需要对列表中的每一种语言执行一些操作”。根据这个道理，`groovy.sql.Sql` 对象提供了一个 `eachRow()` 方法，如清单 17 所示：

清单 **17.** `ResultSet` 迭代

```
import groovy.sql.*

def sql = Sql.newInstance(
   "jdbc:derby://localhost:1527/MyDbTest;create=true",
   "username",
   "password",
   "org.apache.derby.jdbc.ClientDriver")

println("grab a specific field")
sql.eachRow("select name from languages"){ row ->
    println row.name
}

println("grab all fields")
sql.eachRow("select * from languages"){ row ->
    println("Name: ${row.name}")
    println("Version: ${row.version}")
    println("URL: ${row.url}\n")
}
```

该脚本的第一行代码实例化了一个新的 `Sql` 对象：设置 JDBC 连接字符串、用户名、密码和 JDBC 驱动器类。这时，可以调用 `eachRow()` 方法，传递 SQL `select` 语句作为一个方法参数。在闭包内部，可以引用列名（`name`、`version`、`url`），就好像实际存在 `getName()`、`getVersion()` 和 `getUrl()` 方法一样。

这显然要比 Java 语言中的等效方法更加清晰。在 Java 中，必须创建单独的 `DriverManager`、`Connection`、`Statement` 和 `JDBCResultSet`，然后必须在嵌套的 `try` / `catch` / `finally` 块中将它们全部清除。

对于 `Sql` 对象，您会认为 `each()` 或 `eachRow()` 都是一个合理的方法名。但是在接下来的示例中，我想您会认为 `each()` 这个名称并不能充分表达它的功能。

文件迭代

我从未想过使用原始的 Java 代码逐行遍历 `java.io.File`。当我完成了所有的嵌套的 `BufferedReader` 和 `FileReader` 后（更别提每个流程末尾的所有异常处理），我已经忘记最初的目的是什么。

清单 18 展示了使用 Java 语言完成的整个过程：

清单 **18. Java** 文件迭代

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class WalkFile {
   public static void main(String[] args) {
      BufferedReader br = null;
      try {
         br = new BufferedReader(new FileReader("languages.txt"));
         String line = null;
         while((line = br.readLine()) != null) {
            System.out.println("I know " + line);
         }
      }
      catch(FileNotFoundException e) {
         e.printStackTrace();
      }
      catch(IOException e) {
         e.printStackTrace();
      }
      finally {
         if(br != null) {
            try {
               br.close();
            }
            catch(IOException e) {
               e.printStackTrace();
            }
         }
      }
   }
}
```

清单 19 展示了 Groovy 中的等效过程：

清单 **19. Groovy** 文件迭代

```
def f = new File("languages.txt")
f.eachLine{language->
  println "I know ${language}"
}
```

这正是 Groovy 的简洁性真正擅长的方面。现在，我希望您了解为什么我将 Groovy 称为 “Java 程序员的 DSL”。

注意，我在 Groovy 和 Java 语言中同时处理同一个 `java.io.File` 类。如果该文件不存在，那么 Groovy 代码将抛出和 Java 代码相同的 `FileNotFoundException` 异常。区别在于，Groovy 没有已检测的异常。在 `try` / `catch` / `finally` 块中封装 `eachLine()` 结构是我自己的爱好 — 而不是一项语言需求。对于一个简单的命令行脚本中，我欣赏 清单 19 中的代码的简洁性。如果我在运行应用服务的同时执行相同的迭代，我不能对这些异常坐视不管。我将在与 Java 版本相同的 `try/catch` 块中封装 `eachLine()` 块。

`File` 类对 `each()` 方法进行了一些修改。其中之一就是 `splitEachLine(String separator, Closure closure)`。这意味着您不仅可以逐行遍历文件，同时还可以将它分为不同的标记。清单 20 展示了一个例子：

清单 **20.** 分解文件的每一行

```
// languages.txt
// notice the space between the language and the version
Java 1.5
Groovy 1.6
JavaScript 1.x

// splitTest.groovy
def f = new File("languages.txt")
f.splitEachLine(" "){words->
  words.each{ println it }
}

// output
Java
1.5
Groovy
1.6
JavaScript
1.x
```

如果处理的是二进制文件，Groovy 还提供了一个 `eachByte()` 方法。

当然，Java 语言中的 `File` 并不总是一个文件 — 有时是一个目录。Groovy 还提供了一些 `each()` 修改以处理子目录。

目录迭代

使用 Groovy 代替 shell 脚本（或批处理脚本）非常容易，因为您能够方便地访问文件系统。要获得当前目录的目录列表，参见清单 21：

清单 **21.** 目录迭代

```
def dir = new File(".")
dir.eachFile{file->
  println file
}
```

`eachFile()` 方法同时返回了文件和子目录。使用 Java 语言的 `isFile()` 和 `isDirectory()` 方法，可以完成更复杂的事情。清单 22 展示了一个例子：

清单 **22.** 分离文件和目录

```
def dir = new File(".")
dir.eachFile{file->
  if(file.isFile()){
    println "FILE: ${file}"
  }else if(file.isDirectory()){
    println "DIR:  ${file}"
  }else{
    println "Uh, I'm not sure what it is..."
  }
}
```

由于两种 Java 方法都返回 `boolean` 值，可以在代码中添加一个 Java 三元操作符。清单 23 展示了一个例子：

清单 **23.** 三元操作符

```
def dir = new File(".")
dir.eachFile{file->
  println file.isDirectory() ? "DIR:  ${file}" : "FILE: ${file}"
}
```

如果只对目录有兴趣，那么可以使用 `eachDir()` 而不是 `eachFile()` 。还提供了 `eachDirMatch()` 和 `eachDirRecurse()` 方法。

可以看到，对 `File` 仅使用 `each()` 方法并不能提供足够的含义。典型 `each()` 方法的语义保存在 `File` 中，但是方法名更具有描述性，从而提供更多有关这个高级功能的信息。

URL 迭代

理解了如何遍历 `File` 后，可以使用相同的原则遍历 HTTP 请求的响应。Groovy 为 `java.net.URL` 提供了一个方便的（和熟悉的） `eachLine()` 方法。

例如，清单 24 将逐行遍历 ibm.com 主页的 HTML：

清单 **24. URL** 迭代

```
def url = new URL("http://www.ibm.com")
url.eachLine{line->
  println line
}
```

当然，如果这就是您的目的的话，Groovy 提供了一个只包含一行代码的解决办法，这主要归功于 `toURL()` 方法，它被添加到所有 `Strings` ： `"http://www.ibm.com".toURL().eachLine{ println it }` 。

但是，如果希望对 HTTP 响应执行一些更有用的操作，该怎么办呢？具体来讲，如果发出的请求指向一个 RESTful Web 服务，而该服务包含您要解析的 XML，该怎么做呢？ `each()` 方法将在这种情况下提供帮助。

XML 迭代

您已经了解了如何对文件和 URL 使用 `eachLine()` 方法。XML 给出了一个稍微有些不同的问题 — 与逐行遍历 XML 文档相比，您可能更希望对逐个元素进行遍历。

例如，假设您的语言列表存储在名为 languages.xml 的文件中，如清单 25 所示：

清单 **25. languages.xml** 文件

```
<langs>
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
```

Groovy 提供了一个 `each()` 方法，但是需要做一些修改。如果使用名为 `XmlSlurper` 的原生 Groovy 类解析 XML，那么可以使用 `each()` 遍历元素。参见清单 26 所示的例子：

清单 **26. XML** 迭代

```
def langs = new XmlSlurper().parse("languages.xml")
langs.language.each{
  println it
}

//output
Java
Groovy
JavaScript
```

`langs.language.each` 语句从名为 `<language>` 的 `<langs>` 提取所有元素。如果同时拥有 `<format>` 和 `<server>` 元素，它们将不会出现在 `each()` 方法的输出中。

如果觉得这还不够的话，那么假设这个 XML 是通过一个 RESTful Web 服务的形式获得，而不是文件系统中的文件。使用一个 URL 替换文件的路径，其余代码仍然保持不变，如清单 27 所示：

清单 **27. Web** 服务调用的 **XML** 迭代

```
def langs = new XmlSlurper().parse("http://somewhere.com/languages'
langs.language.each{
  println it
}
```

这真是个好方法，`each()` 方法在这里用得很好，不是吗？

结束语

在使用 `each()` 方法的整个过程中，最妙的部分在于它只需要很少的工作就可以处理大量 Groovy 内容。解了 `each()` 方法之后，Groovy 中的迭代就易如反掌了。正如 Raymond 所说，这正是关键所在。一旦了解了如何遍历 `List` ，那么很快就会掌握如何遍历数组、`Map` 、`String` 、`Range` 、`enum` 、SQL `ResultSet` 、`File` 、目录和 `URL` ，甚至是 XML 文档的元素。

本文的最后一个示例简单提到使用 `XmlSlurper` 实现 XML 解析。在下一期文章中，我将继续讨论这个问题，并展示使用 Groovy 进行 XML 解析有多么简单！您将看到 `XmlParser` 和 `XmlSlurper` 的实际使用，并更好地了解 Groovy 为什么提供两个类似但又略有不同的类实现 XML 解析。到那时，希望您能发现 Groovy 的更多实际应用。

下载

描述	名字	大小
本文源代码	j-pg04149.zip	17KB

实战 Groovy: Groovy：Java 程序员的 DSL

用 *Groovy* 编写更少的代码，完成更多的工作

Groovy 专家 Scott Davis 将重新开始撰写 实战 *Groovy* 系列文章，该系列文章于 2006 年停止编写。作为开篇文章，本文将介绍 Groovy 最近的发展以及 Groovy 当前的状态。然后了解大约 从 2009 年开始，使用 Groovy 是多么轻松。

Andrew Glover 于 2004 年开始为 developerWorks 撰写关于 Groovy 的文章，他先撰写了 *alt.lang.jre* 系列中的介绍性文章 “alt.lang.jre: 感受 Groovy”，又继续撰写了长期刊发的 实战 *Groovy* 系列。发表这些文章时市场上还没有出现关于 Groovy 的书籍（现在这样的书籍超过十几本），而且 Groovy 1.0 在几年后才于 2007 年 1 月发布。自 2006 年末发布 实战 *Groovy* 的最后一期后，Groovy 发生了很大的变化。

现在，Groovy 每个月的平均下载数量大约为 35,000。Mutual of Omaha 等保守的公司拥有超过 70,000 行 Groovy 生产代码。Groovy 在 Codehaus.org 中有一个最忙碌的邮件列表，这是托管该项目的位置（请参阅 参考资料）。Grails 是惟一一个拥有较多下载数量以及繁忙邮件列表的项目，它是在 Groovy 中实现的流行 Web 框架（请参阅 参考资料）。

在 JVM 中运行非 Java™ 语言不仅常见，而且也是 Sun 的 JVM 策略的核心部分。Groovy 加入进了 Sun 支持的备选语言（如 JavaScript, JavaFX, JRuby 和 Jython）行列中。2004 年所做的实验现在成为了最前沿的技术。

2009 年撰写的关于 Groovy 的文章在许多方面与 Andy 开始撰写的文章相同。2005 年确立的语法仍然保留至今。每个发行版都添加了引人注目的新功能，但是对于项目主管来说，保留向后兼容性是极为重要的。这项可靠的基础使得 Java 开发组织在其应用程序进入生产环境并开始依赖各项技术时，毫不犹豫地选择了 Groovy。

本文的目标是使经验丰富的 Java 开发人员可以像 Groovy 开发人员一样进行快速开发。不要被它的表面所蒙骗。本系列如名称所示全部都是实际使用的 Groovy 实践知识。在最开始编写完 “Hello, World” 之后，请准备好尽管掌握实际的应用。

关于本系列

Groovy 是运行在 Java 平台上的现代编程语言。它将提供与现有 Java 代码的无缝集成，同时引入闭包和元编程等出色的新功能。简言之，Groovy 是 21 世纪根据 Java 语言的需要编写的。

把任意一个新工具集成到开发工具包中的关键是：知道何时使用以及何时不应使用该工具。Groovy 可以提供强大的功能，但是必须正确地应用到适当的场景中。为此，实战 *Groovy* 系列将探究 Groovy 的实际应用，帮助您了解何时及如何成功应用它们。

安装 Groovy

如果您以前从未使用过 Groovy，则首先需要安装 Groovy。安装步骤非常简单，这些步骤与安装 Ant 和 Tomcat 等常见 Java 应用程序甚至安装 Java 平台本身的步骤相同：

1. 下载 最新的 Groovy ZIP 文件或 tarball。
2. 将存档解压缩到所选目录中（您应当避免在目录名称中使用空格）。
3. 创建 `GROOVY_HOME` 环境变量。
4. 把 GROOVY_HOME/bin 添加到 `PATH` 中。

Groovy 运行在 Java 5 或 6 上的效果最佳。在命令提示中输入 `java -version` 以确认您使用的是最新版本。然后键入 `groovy -version` 以确保 Groovy 已正确安装。

所有主要 IDE（Eclipse、IntelliJ 和 NetBeans）都有支持自动完成和分步调试等功能的 Groovy 插件。虽然拥有一个优秀的 IDE 对于目前编写 Java 代码来说几乎成为了一项必要要求，但是对于 Groovy 来说并非绝对。得益于 Groovy 语言的简明性，许多人都选择使用简单的文本编辑器编写。vi 和 Emacs 等常见开源编辑器都提供 Groovy 支持，Textpad（面向 Windows®）和 TextMate（面向 Mac OS X）等便宜的商业文本编辑器也提供 Groovy 支持（有关更多信息，请参阅 参考资料）。

您稍后将在本文中看到，将 Groovy 与现有 Java 项目集成起来十分简单。您只需将一个 Groovy JAR 从 GROOVY_HOME/embeddable 添加到类路径中并把现有的 `javac` Ant 任务封装到 `groovyc` 任务中（Maven 提供类似的支持）。

但是在掌握这些知识之前，我将从必修的 “Hello World” 示例开始。

进入 Groovy 世界

您知道 “Hello World” 示例应当会演示哪些内容 — 它是用给定语言可以编写的最简单的程序。清单 1 中所示的 “Hello World” Java 代码的有趣之处在于，需要了解中间语言的知识才能完全了解代码含义：

清单 **1.** 用 **Java** 代码编写的 **“Hello World”** 示例

```
public class HelloJavaWorld{
  public static void main(String[] args){
    System.out.println("Hello Java World");
  }
}
```

首先创建名为 HelloJavaWorld.java 的文件并输入 `public class HelloJavaWorld`。许多刚开始使用 Java 的开发人员学到的第一课是如果类名与文件名不完全匹配（包括大小写），则类无法编译。另外，好奇的学生将在此时开始询问关于 `public` 和 `private` 之类的访问修饰符。

下一行 — `public static void main(String[] args)` — 通常将引发关于实现细节的一连串问题：什么是 `static` ？什么是 `void` ？为什么需要将方法命名为 `main` ？什么是 `String` 数组？而最后，尝试向刚开始使用 Java 的开发人员说明 `out` 是 `System` 类中的 `PrintStream` 对象的 `public` 、 `static` 、 `final` 实例。我永远也忘不了学生说“天哪！其实我只是想输出 ‘Hello’”。

将此示例与用 Groovy 编写的 “Hello World” 进行对照。创建名为 HelloGroovyWorld.groovy 的文件并输入清单 2 中所示的代码行：

清单 **2.** 用 **Groovy** 代码编写的 **“Hello World”** 示例

```
println "Hello Groovy World"
```

是的，这段代码是与清单 1 中所示的 Java 示例等效的 Groovy 代码。在本例中，所有实现细节 — 并不立即解决手头问题的“知识” — 都隐藏在后台，只显示简单输出 “Hello” 的代码。输入 `groovy HelloGroovyWorld` 以确认它可以工作。

这个小示例将演示 Groovy 的双重价值：它将显著地减少需要编写的代码行数，同时保留 Java 等效代码的语义。在下一节中，您将进一步探究这种理念。

深入研究 Hello World

有经验的 Java 开发人员都知道在 JVM 中运行代码之前必须先编译这些代码。但是，Groovy 脚本在任何位置都不显示为类文件。这是否意味着可以直接执行 Groovy 源代码？答案是“不一定，但是它看上去是这样，对不对？”

Groovy 解释器将先在内存中编译源代码，然后再将其转到 JVM 中。您可以通过输入 `groovyc HelloGroovyWorld.groovy` 手动执行此步骤。但是，如果尝试使用 `java` 运行得到的类，将显示清单 3 中所示的异常：

清单 **3.** 尝试在 **`CLASSPATH`** 中没有 **Groovy JAR** 的情况下运行经过编译的 **Groovy** 类

```
$ java HelloGroovyWorld
Exception in thread "main" java.lang.NoClassDefFoundError: groovy/l
```

如前述，Groovy JAR 必须包含在 `CLASSPATH` 中。再次尝试执行代码，这一次把 `-classpath` 实参传递到 `java` 中，如清单 4 所示：

清单 **4.** 用 **`java`** 命令成功运行经过编译的 **Groovy** 类

```
//For UNIX, Linux, and Mac OS X
$ java -classpath $GROOVY_HOME/embeddable/groovy-all-x.y.z.jar:. He
Hello Groovy World

//For Windows
$ java -classpath %GROOVY_HOME%/embeddable/groovy-all-x.y.z.jar;. H
Hello Groovy World
```

现在您已经取得了一些进展。但是为了证明 Groovy 脚本真正地保留 Java 示例的语义，您需要深入钻研字节码。首先，输入 `javap HelloJavaWorld` ，如清单 5 所示：

清单 **5.** 解释 **Java** 字节码

```
$ javap HelloJavaWorld
Compiled from "HelloJavaWorld.java"
public class HelloJavaWorld extends java.lang.Object{
  public HelloJavaWorld();
  public static void main(java.lang.String[]);
}
```

除了为您添加了 `javac` 编译器之外，这段代码中不应当有过多令人惊讶之处。您无需显式输入 `extends java.lang.Object` 或提供类的默认构造函数。

现在，输入 `javap HelloGroovyWorld` ，如清单 6 所示：

清单 **6.** 解释 **Groovy** 字节码

```
$ javap HelloGroovyWorld
Compiled from "HelloGroovyWorld.groovy"
public class HelloGroovyWorld extends groovy.lang.Script{
  ...
  public static void main(java.lang.String[]);
  ...
}
```

什么是 **DSL**？

Martin Fowler 普及了特定于领域语言的理念（请参阅 参考资料）。他把 DSL 定义为“侧重特定领域的表达有限的计算机编程语言”。“有限的表达”并不是指语言的用途有限，只是表示这种语言提供了足够用于适当表达“特定领域”的词汇表。DSL 是一种很小的专用语言，这与 Java 语言等大型通用语言形成对比。

SQL 就是一种优秀的 DSL。您无法使用 SQL 编写操作系统，但它是处理关系数据库这一有限领域的理想选择。在同样意义上，Groovy 是 Java 平台的 DSL，因为它是有限领域的 Java 开发的理想选择。我在这里使用 *DSL* 是为了启发读者，并不是特别的精确。如果我把 Groovy 称为 常用 *Java* 语言的内部 *DSL*，可能更容易被 DSL 纯粹主义者接受。

Dave Thomas 进一步阐明 DSL 的概念（请参阅 参考资料）。他写道，“无论领域专家在何时交流……他们都在说行业术语，这是他们为与同行进行有效交流而创造出的更简略的专用语言”。可能将 Groovy 视为“简略的 Java 语言”更能说明 Groovy 与 Java 语言之间的关系。本文的下一节将提供另一个示例。

在这里，您可以看到 `groovyc` 编译器将获取源文件的名称并创建了一个同名的类（该类扩展 `groovy.lang.Script` 而非 `java.lang.Object` ，这应当能帮助您理解尝试在 CLASSPATH 中没有 Groovy JAR 的情况下运行文件抛出 `NoClassDefFoundError` 异常的原因）。在所有其他编译器提供的方法之中，您应当能够找到一种优秀的旧 `public static void main(String[] args)` 方法。 `groovyc` 编译器把脚本行封装到此方法中以保留 Java 语义。这意味着在使用 Groovy 时可以利用所有的现有 Java 知识。

例如，下面是在 Groovy 脚本中接受命令行输入的方法。创建名为 Hello.groovy 的新文件并添加清单 7 中的代码行：

清单 **7.** 接受命令行输入的 **Groovy** 脚本

```
println "Hello, " + args[0]
```

现在在命令行中输入 `groovy Hello Jane` 。 `args``String` 数组就在这里，就像任何一位 Java 开发人员期望的那样。在这里使用 `args` 对于新手可能没意义，但是它对于经验丰富的 Java 开发人员意义非凡。

Groovy 将把 Java 代码缩减为基本要素。您刚刚编写的 Groovy 脚本几乎和可执行的伪代码一样。表面上，该脚本简单得足以让新手能够理解，但是对于经验丰富的开发人员，它没有去掉 Java 语言的底层强大功能。这就是我将 Groovy 视为 Java 平台的特定于领域语言（DSL）的原因（请参阅 什么是 DSL？ 侧栏）。

普通的旧 **Groovy** 对象

JavaBean — 或更通俗的名称，普通的旧 Java 对象（Plain Old Java Object，POJO）— 是 Java 开发的主要支柱。在创建 POJO 以表示域对象时，您应当遵循定义好的一组期望。类应当为 `public` ，并且字段应当为带有一组对应的 `public` getter 和 setter 方法的 `private` 。清单 8 显示了一个典型的 Java POJO：

清单 **8. Java POJO**

```
public class JavaPerson{
  private String firstName;
  private String lastName;

  public String getFirstName(){ return firstName; }
  public void setFirstName(String firstName){ this.firstName = firs

  public String getLastName(){ return lastName; }
  public void setLastName(String lastName){ this.lastName = lastNar
}
```

普通的旧 Groovy 对象（Plain Old Groovy Object，POGO）是 POJO 的简化的替代者。它们完全保留了 POJO 的语义，同时显著减少了需要编写的代码量。清单 9 显示了用 Groovy 编写的 “简易” person 类：

清单 9. Groovy POGO

```
class GroovyPerson{
  String firstName
  String lastName
}
```

除非您另外指定，否则 Groovy 中的所有类都是 `public` 的。所有属性都是 `private` 的，而所有方法都是 `public` 的。编译器将为每个属性自动提供一组 `public` getter 和 setter 方法。用 `javac` 编译 `JavaPerson` 并用 `groovyc` 编译 `GroovyPerson`。现在通过 `javap` 运行它们以确认该 Groovy 示例拥有 Java 示例所拥有的所有内容，甚至可以扩展 `java.lang.Object`（在先前的 `HelloGroovyWorld` 示例中未指定类，因此 Groovy 转而创建了扩展 `groovy.lang.Script` 的类）。

所有这些意味着您可以立即开始使用 POGO 作为 POJO 的替代选择。Groovy 类是只剩下基本元素的 Java 类。在编译了 Groovy 类之后，其他 Java 类可以轻松地使用它，就好像它是用 Java 代码编写的。为了证明这一点，请创建一个名为 JavaTest.java 的文件并添加清单 10 中的代码：

清单 10. 从 Java 代码中调用 Groovy 类

```
public class JavaTest{
  public static void main(String[] args){
    JavaPerson jp = new JavaPerson();
    jp.setFirstName("John");
    jp.setLastName("Doe");
    System.out.println("Hello " + jp.getFirstName());

    GroovyPerson gp = new GroovyPerson();
    gp.setFirstName("Jane");
    gp.setLastName("Smith");
    System.out.println("Hello " + gp.getFirstName());
  }
}
```

即使 getter 和 setter 不显示在 Groovy 源代码中，这项测试也证明了它们是存在于经过编译的 Groovy 类中并且完全可以正常运行。但是如果我不向您展示 Groovy 中的相应测试，则这个示例不算完整。创建一个名为 TestGroovy.groovy 的文件并添加清单 11 中的代码：

清单 **11.** 从 **Groovy** 中调用 **Java** 类

```
JavaPerson jp = new JavaPerson(firstName:"John", lastName:"Doe")
println "Greetings, " + jp.getFirstName() + ".
   It is a pleasure to make your acquaintance."

GroovyPerson gp = new GroovyPerson(lastName:"Smith", firstName:"Jar
println "Howdy, ${gp.firstName}. How the heck are you?"
```

您首先可能会注意到，Groovy 提供的新构造函数允许您命名字段并按照所需顺序指定这些字段。甚至更有趣的是，可以在 Java 类或 Groovy 类中使用此构造函数。这怎么可能？事实上，Groovy 先调用默认的无实参构造函数，然后调用每个字段的相应 setter。您可以模拟 Java 语言中的类似行为，但是由于 Java 语言缺少命名实参并且两个字段都是 `Strings` ，因此您无法按照任何顺序传入名字和姓氏字段。

接下来，注意 Groovy 支持传统的 Java 方法来执行 `String` 串联，以及在 `String` 中直接嵌入用 `${}` 圈起的代码的 Groovy 方法（这些称为 `GString` ，它是 *Groovy Strings* 的简写）。

最后，在类中调用 getter 时，您将看到使用 Groovy 语法的更多优点。您无需使用更冗长的 `gp.getFirstName()` ，只需调用 `gp.firstName` 。看上去您是在直接访问字段，但是事实上您在调用后台的相应的 getter 方法。Setter 将按照相同的方法工作： `gp.setLastName("Jones")` 和 `gp.lastName = "Jones"` 是等效的，后者将在后台调用前者。

我希望您也能够承认，在任何情况下，Groovy 都类似于简易版的 Java 语言 —"领域专家" 可能使用 Groovy "与同行进行有效地交流"，或者类似于老朋友之间随意的谈笑。

归根结底，Groovy 就是 Java 代码

Groovy 最被低估的一个方面是它完全支持 Java 语法的事实。如前述，在使用 Groovy 时，您不必放弃一部分 Java 知识。在开始使用 Groovy 时，大部分代码最终看上去都像是传统的 Java 代码。但是随着您越来越熟悉新语法，代码将逐渐发展为包含更简明、更有表现力的 Groovy 风格。

要证明 Groovy 代码可以看上去与 Java 代码完全相同，请将 JavaTest.java 复制到 JavaTestInGroovy.groovy 中，然后输入 `groovy JavaTestInGroovy`。您应当会看到同样的输出，但是请注意，您无需在运行前先编译 Groovy 类。

此次演示应当使经验丰富的 Java 开发人员能够不假思索地选择 Groovy。由于 Java 语法也是有效的 Groovy 语法，因此最开始的学习曲线实际上是不存在的。您可以将现有的 Java 版本与 Groovy、现有的 IDE 以及现有的生产环境结合使用。这意味着对您日常工作的干扰非常少。您只需确保 Groovy JAR 位于 `CLASSPATH` 中并调整构建脚本，以便 Groovy 类与 Java 类同时编译。下一节将向您展示如何向 Ant build.xml 文件中添加 `groovyc` 任务。

用 Ant 编译 Groovy 代码

如果 `javac` 是可插拔的编译器，则可以指示它同时编译 Groovy 和 Java 文件。但是它不是，因此您只需用 `groovyc` 任务在 Ant 中封装 `javac` 任务。这将允许 `groovyc` 编译 Groovy 源代码，并允许 `javac` 编译 Java 源代码。

当然，`groovyc` 既可以编译 Java 文件，又可以编译 Groovy 文件，但是还记得 `groovyc` 添加到 `HelloGroovyWorld` 和 `GroovyPerson` 中的额外的方便方法么？这些额外的方法也将被添加到 Java 类中。最好的方法可能就是让 `groovyc` 编译 Groovy 文件，而让 `javac` 编译 Java 文件。

要从 Ant 中调用 `groovyc`，请使用 `taskdef` 定义任务，然后像平时使用 `javac` 任务一样使用 `groovyc` 任务（有关更多信息，请参阅 参考资料）。清单 12 显示了 Ant 构建脚本：

清单 12. 用 Ant 编译 Groovy 和 Java 代码

```
<taskdef name="groovyc"
         classname="org.codehaus.groovy.ant.Groovyc"
         classpathref="my.classpath"/>

<groovyc srcdir="${testSourceDirectory}" destdir="${testClassesDire
 <classpath>
   <pathelement path="${mainClassesDirectory}"/>
   <pathelement path="${testClassesDirectory}"/>
   <path refid="testPath"/>
 </classpath>
 <javac debug="on" />
</groovyc>
```

顺便说一下，包含 `${}` 的 `String` 看上去疑似 `GString`，是不是？Groovy 是一种优秀的语言，它从各种其他语言和库中借鉴了语法和功能。您经常会觉得似乎在其他语言中见过类似的特性。

结束语

这是一次 Groovy 旋风之旅。您了解了一些关于 Groovy 曾经的地位及其目前现状的信息。您在系统中安装了 Groovy，并且通过一些简单的示例，了解了 Groovy 为 Java 开发人员提供的强大功能。

Groovy 不是运行在 JVM 上的惟一的备选语言。JRuby 是了解 Ruby 的 Java 开发人员的优秀解决方案。Jython 是了解 Python 的 Java 开发人员的优秀解决方案。但是正如您所见，Groovy 是了解 Java 语言的 Java 开发人员的优秀解决方案。Groovy 提供了类似 Java 的简明语法，同时保留了 Java 语义，这一点非常引人注目。而且，采用新语言的路径不包含 `del *.*` 或 `rm -Rf *` 是一次很好的变革，您不这样认为吗？

下期文章中，您将了解 Groovy 中的迭代。代码通常需要逐项遍历内容，不管它是列表、文件，还是 XML 文档。您将看到非常普遍的 `each` 闭包。到那时，我希望您可以发现大量 Groovy 的实际应用。

实战 Groovy: 关于 MOP 和迷你语言

Groovy 让元对象协议从实验室走进应用程序

将耳朵贴到地上仔细听 —— MOP 正在前进！了解一下元对象协议（Meta Object Protocol，MOP）吧，这是一种将应用程序、语言和应用程序构建 为语言的翻新方法。

在最近的一次采访中，Groovy 项目经理 Guillaume Laforge 提到，他最喜欢的 Groovy 特性是它实现了 元对象协议（*Meta Object Protocol*）或称 MOP。在运行时向一个对象传递方法，或者 消息时，这个协议使对象可以作出影响它自己的状态或者行为的特定选择。正如在 PARC Software Design Area 项目主页上所描述的（请参阅 参考资料）：

> 元对象协议方法……基于这样一种想法，即人们可以并且应当使语言开放，使用户可以根据他们的特定需要调整设计和实现。换句话说，鼓励用户参与语言设计过程。

显然，这种大胆的思路提出了构建更智能的应用程序、甚至 语言的令人激动的可能性。在本月的专栏中，我将展示 Groovy 是如何实现 MOP 的，然后用一个实际的例子介绍它最令人激动的实际应用：作为一种迷你语言的字典！

在 下载一节中提供了字典应用程序示例的源代码。下面的例子中需要下载 DbUnit，请参阅 参考资料。

关于本系列

在自己的开发工作中引入任何工具的关键是知道什么时候使用它、什么时候将它放回工具箱。脚本（或者 动态）语言会为您的工具箱加入特别强大的工具，前提是在适合的场景正确地使用它。在这方面，实战 *Groovy*是专门探讨 Groovy 的实际使用、并教给您什么时候以及如何成功地使用它们的系列文章。

魔术般的 MOP

元对象协议不是 Groovy 独有的，它也不是由 Groovy 发明者发明的。事实上，它的身世可追溯到 LISP 和 AOP 背后的一些人。考虑到这种身世，MOP 受到 Groovy 的见多识广的创造者们的欢迎就毫不奇怪了。

在 Groovy 中可以实现 MOP 是因为，在 *Groovyland*中每一个对象都隐式地实现 `groovy.lang.GroovyObject` ，它定义了 `invokeMethod()` 和 `getProperty()` 两个方法。在运行时，如果向一个对象传递消息，而这个对象不是作为类或者其子类中定义的属性或者方法存在，那么就调用 `getProperty()` 或者 `invokeMethod()` 方法。

在清单 1 中，我定义了 Groovy 类 `MOPHandler` ，它实现了 `invokeMethod()` 和 `getProperty()` 。当创建了 `MOPHandler` 的一个实例后，可以调用任意数量的方法或者属性，并看到它打印出说明调用了什么的消息。

清单 **1. MOP** 得到调用的句柄

```
class MOPHandler {

  def invokeMethod(String method, Object params) {
    println "MOPHandler was asked to invoke ${method}"
    if(params != null){
     params.each{ println "\twith parameter ${it}" }
    }
  }

  def getProperty(String property){
     println "MOPHandler was asked for property ${property}"
  }
}
def hndler = new MOPHandler()
hndler.helloWorld()
hndler.createUser("Joe", 18, new Date())
hndler.name
```

继续，运行清单 1 中的代码，就会看到如清单 2 所示的输出。

清单 **2.** 您不相信我，是不是？

```
aglover@glove-ubutu:~/projects/groovy-mop$ groovy
 ./src/groovy/com/vanward/groovy/MOPHandler1.groovy
MOPHandler was asked to invoke helloWorld
MOPHandler was asked to invoke createUser
       with parameter Joe
       with parameter 18
       with parameter Sun Sep 04 10:32:22 EDT 2005
MOPHandler was asked for property name
```

是不是很不错？ MOP 就像一个安全网，捕捉传递错误的消息，但是这不是它最妙的功能。用它创建可以以 一般的方式智能地对传递进来的任何消息作出响应的智能对象，才是它的最大亮点。

让我成为一种迷你语言

用 MOP 可以做的一件有趣的事情是创建伪领域专用语言，或者称为 迷你语言。这些是专门用于解决特定问题的独特语言。与像 Java、C# 或者甚至 Groovy 这样的流行语言不同，它们被看成是用来解决任何问题的一般性语言，而迷你语言只针对特定问题。需要一个例子？请考虑 Unix 及其 shell，如 Bash。

只是为了练习 —— 并且这样可以真正 _感受_MOP —— 我将在本文的其余部分创建一个字典应用程序，它本身实际就是一个迷你语言。这个应用程序将提供一个查询字典的界面。它让用户可以创建新的单词项、得到给定单词的定义、得到给定单词的同义字、查询单词的词性以及删除单词。表 1 总结了这个迷你语言。

表 **1.** 字典迷你语言的语义

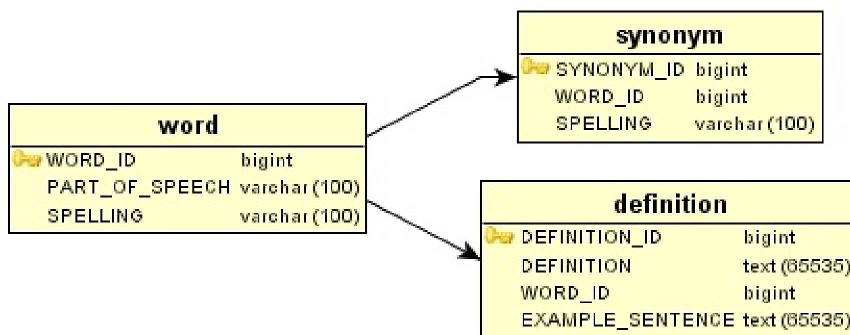
这个字典应用程序语言有以下语义（按从最特殊到最一般排列）：

1. 词性	要查询单词的词性，消息应该以 `is` 打头，后跟这个单词，然后是 `a` 或者 `an` ，然后是词性。
2. 同义词	要查询一个单词的同义词，消息应该以 `synonymsOf` 打头，然后是这个单词。
3. 删除单词	要从字典中删除一个单词，消息应该以 `remove` 或 `delete` 打头，然后是这个单词。
4. 创建单词	要在字典中创建一个新单词，可将单词作为方法，并将其定义、词性和可选的一组同义词作为参数传递。
5. 获取定义	要查询一个单词的定义，可将这个单词作为属性或者方法传递。

字典应用程序

字典应用程序基于表结构如图 1 所示的数据库。如果您经常读我们的文章，那么您可能会看出这个表结构源自专栏文章“Mark it up with Groovy Builders”。

图 **1.** 字典的简单数据库模型



注意：我将忽略 `definition` 的 `EXAMPLE_SENTENCE` 这一列。

我将创建一个简单的 facade，它将作为用户与数据库之间的接口。这个 facade 将通过提供 `invokeMethod` 和 `getProperty` 的实现来利用 Groovy 的 MOP 特性。`invokeMethod` 方法将确定命令并将责任委派给相应的内部 `private` 方法。

一些前提条件

因为这个应用程序依赖于数据库，在继续之前，可能需要复习一下 GroovySql。我还会使用 Groovy 的正则表达式（在 感受 Groovy中介绍）来确定传递给 facade 的消息。

我会在写这个 facade 时，_用_Groovy 测试它，因此需要回顾 Groovy 的单元测试能力，请参阅我的“Unit test your Java code faster with Groovy”。

最后，在测试过程中，我将用 DbUnit 管理数据库状态。因为在本系列中没有写过 DbUnit 的内容，我将在继续之前简单介绍在 Groovy 中使用 DbUnit。

DbUnit，结合 Groovy

DbUnit 是一项 JUnit 扩展，它在每一次运行测试之前，使数据库处于一种已知状态。在 Groovy 中使用 DbUnit 非常简单，因为 DbUnit 提供了一个 API，可以在测试用例中进行委派。为了使用 DbUnit，需要向它提供一个数据库连接和一个包含作为数据库种子的文件。将它们插入到 JUnit 的 fixture 机制中（即 `setUp()` ）后，就可以继续了！ 清单 3 显示了字典应用程序的开始测试类。

清单 **3.** 字典应用程序的开始测试类

```
package test.com.vanward.groovy
import com.vanward.groovy.SimpleDictionary
import groovy.util.GroovyTestCase
import java.io.File
import java.sql.Connection
import java.sql.DriverManager
import org.dbunit.database.DatabaseConnection
import org.dbunit.database.IDatabaseConnection
import org.dbunit.dataset.IDataSet
import org.dbunit.dataset.xml.FlatXmlDataSet
import org.dbunit.operation.DatabaseOperation
class DictionaryTest extends GroovyTestCase{
  def dictionary

  void setUp() {
     this.handleSetUpOperation()
     dictionary = new SimpleDictionary()
  }

  def handleSetUpOperation() {
     def conn = this.getConnection()
     def data = this.getDataSet()
     try{
         DatabaseOperation.CLEAN_INSERT.execute(conn, data)
     }finally{
         conn.close()
     }
   }

   def getDataSet()  {
     return new FlatXmlDataSet(new File("test/conf/words-seed.xml'
   }

   def getConnection()  {
      Class.forName("org.gjt.mm.mysql.Driver")
      def jdbcConnection = DriverManager.
            getConnection("jdbc:mysql://localhost/words",
                   "words", "words")
      return new DatabaseConnection(jdbcConnection)
   }
}
```

在清单 3 中，我创建了类 shell，它作为在编写字典应用程序时的测试类。调用测试时，JUnit 会调用 `setUp()` ，它又会调用 DbUnit 的 API。DbUnit 会将在文件 test/conf/words-seed.xml 中找到的数据插入到数据库中。文件 test/conf/words-seed.xml 的内容可见清单 4。

清单 **4.** 示例种子文件

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
<word WORD_ID="1" SPELLING="pugnacious" PART_OF_SPEECH="Adjective'
<definition DEFINITION_ID="10"
            DEFINITION="Combative in nature; belligerent."
            WORD_ID="1" />
<synonym SYNONYM_ID="20" WORD_ID="1" SPELLING="belligerent"/>
<synonym SYNONYM_ID="21" WORD_ID="1" SPELLING="aggressive"/>
<word WORD_ID="2" SPELLING="glib" PART_OF_SPEECH="Adjective"/>
<definition DEFINITION_ID="11"
               DEFINITION="Performed with a natural, offhand ease'
            WORD_ID="2" />
<definition DEFINITION_ID="12"
           DEFINITION="Marked by ease and fluency of speech or
           writing that often suggests or stems from insincerity,
           superficiality, or deceitfulness"
        WORD_ID="2" />
<synonym SYNONYM_ID="30" WORD_ID="2" SPELLING="artful"/>
<synonym SYNONYM_ID="31" WORD_ID="2" SPELLING="suave"/>
<synonym SYNONYM_ID="32" WORD_ID="2" SPELLING="insincere"/>
<synonym SYNONYM_ID="33" WORD_ID="2" SPELLING="urbane"/>

<word WORD_ID="3" SPELLING="rowel" PART_OF_SPEECH="Verb"/>
<definition DEFINITION_ID="50"
            DEFINITION="to vec, trouble"
            WORD_ID="13" />
</dataset>
```

种子文件的 XML 元素（如清单 4 所示）匹配表名。元素的属性匹配对应的表列。

构建一种迷你语言

定义了测试类后，就可以开始开发（并测试）这个应用程序了。我将按 表 1的排列顺序处理每一项特性。

正则表达式组

正则表达式组在字典例子中起了主要作用。在 Groovy 中，可以通过 `=~` 语法创建一个普通 Java `Matcher` 实例。可以用 `Matcher` 实例，通过调用 `group()` 方法获得 `String` 代码段。使用括号在一个正则表达式中创建组。例如，正则表达式 `(synonymsOf)(.*)` 创建两个组。一个组完全匹配 `String` *"synonymsOf"*，而另一个匹配 *"synonymsOf"*后面的 任何字符。不过要记住，在要获得组值之前，首先必须调用 `Matcher` 的 `matches()` 方法。

1. 词性

如果用户想要查询一个单词的词性，他或者她可以在消息中传递像 `isRowelAVerb` 或者 `isEstivalAnAdjective` 这样的内容。（注意，我使用了 camel-case 语义，并通过同时允许 `a` 和 `an` 以尽量符合正规的英语。）回答问题的智能逻辑就变成了确定正确的单词，在第一种情况下它是 *Rowel*，而在第二种情况下是 *Estival*。还必须确定词性，在本例中分别为 *Verb*和 *Adjective*。

使用正则表达式，逻辑就变得简单了。模式成为 `is(.*)(An|A)(Verb|Adjective|Adverb|Noun)` 。我只对第一和第三组感兴趣（单词和词性）。有了这些，就可以编写一个简单的数据库查询，以获取词性并比较问题和答案，如清单 5 所示。

清单 **5.** 确定词性

```
private determinePartOfSpeech(question){
 def matcher = question =~ 'is(.*)(An|A)(Verb|Adjective|Adverb|Nou
 matcher.matches()
 def word = matcher.group(1)
 def partOfSpeech = matcher.group(3)
 def row = sql.firstRow("select part_of_speech from word
      where spelling=?", [word])
 return row[0] == partOfSpeech
}
```

清单 5 看起来很简单，但是我为它编写了几个测试用例，看看会是什么情况。

清单 **6.** 测试单词词性的确定

```
void testPartOfSpeechFalse() {
 def val = dictionary.isPugnaciousAVerb()
 assertFalse("pugnacious is not a verb", val)
}
void testPartOfSpeechTrue() {
 def val = dictionary.isPugnaciousAnAdjective()
 assertTrue("pugnacious is an Adjective", val)
}
```

再看看清单 4 中的 XML。因为我用 DbUnit 让数据库在每次测试之前处于一种已知状态，因此可以假定单词 *pugnacious*是有效的，且其词性设置为 *Adjective*。在清单 6 中，我在 `DictionaryTest` 中加入了两个简单的测试（请参阅 清单 3），以确保逻辑正确地工作。

2. 同义词

查询同义词的语义模式如下：`synonymsOfBloviate`，其中所要查的单词（*Bloviate*）跟在 `synonymsOf` 后面。正则表达式更简单：`(synonymsOf)(.*)`。找到所需要的单词后，逻辑就进行一个数据库查询，该查询会连接 `word` 和 `synonym` 表。返回的同义词加入到一个 `List` 中并返回，如清单 7 所示。

清单 **7. getSynonyms** 实现

```
private getSynonyms(question){
 def matcher = question =~ '(synonymsOf)(.*)'
 matcher.matches()
 def word = matcher.group(2).toLowerCase()
 def syns = []
 sql.eachRow("select synonym.spelling from synonym, word " +
     "where synonym.word_id = word.word_id and " +
     "word.spelling = ?", [word]){ arow ->
         syns << arow.spelling
      }
 return syns
}
```

清单 8 中的测试验证了，定义了同义词的单词可以正确地返回同义词，而没有同义词的单词（*Rowel*）返回一个空 `List`。

清单 **8.** 不要忘记测试这个方法！

```
void testSynonymsForWord() {
 def val = dictionary.synonymsOfPugnacious()
 def expect = ["belligerent","aggressive"]
 assertEquals("should be: " + expect, expect, val)
}
void testNoSynonymsForWord() {
 def val = dictionary.synonymsOfRowel()
 def expect = []
 assertEquals("should be: " + expect, expect, val)
}
```

3. 删除单词

清单 9 中的删除逻辑是灵活的，因为我允许使用 `remove` 或者 `delete` 后面加单词的命令。例如，`removeGlib` 和 `deleteGlib` 都是有效的命令。因此正则表达式变成：`(remove|delete)(.*)`，而逻辑是一个 SQL `delete`。

清单 **9.** 删除单词

```
private removeWord(word){
 def matcher = word =~ '(remove|delete)(.*)'
 matcher.matches()
 def wordToRemove = matcher.group(2).toLowerCase()
 sql.execute("delete from word where spelling=?" , [wordToRemove]
}
```

当然，这种灵活意味着我必须为删除单词编写 至少两个测试用例，如清单 10 所示。要验证单词真的被删除了，我要获取它们的定义（更多信息请参阅 5. 获取单词的定义）并确定没有返回任何东西。

清单 **10.** 测试这两种情况

```
void testDeleteWord() {
 dictionary.deleteGlib()
 def val =  dictionary.glib()
 def expect = []
 assertEquals("should be: " + expect, expect, val)
}
void testRemoveWord() {
 dictionary.removePugnacious()
 def val =  dictionary.pugnacious()
 def expect = []
 assertEquals("should be: " + expect, expect, val)
}
```

4. 创建单词

创建一个新单词的语义是一个不匹配任何查询模式并且包含一个参数清单的消息。例如，一个像 `echelon("Noun", ["a level within an organization"])` 这样的消息就符合这种模式。单词是 *echelon*，第一个参数是词性，然后是包含定义的 `List` ，还有可选的第三个参数，它可以是同义词的 `List` 。

如清单 11 所示，在数据库中创建一个新单词就是在正确的表（ `word` 和 `definition` ）中插入这个单词及其定义，而且，如果有同义词 `List` 的话，那么要把它们都要加入。

清单 **11.** 创建一个单词

```
private createWord(word, defsAndSyms){
 def wordId = id++
 def definitionId = wordId + 10
 sql.execute("insert into word
      (word_id, part_of_speech, spelling) values (?, ?, ?)" ,
   [wordId, defsAndSyms[0], word])
 for(definition in defsAndSyms[1]){
   sql.execute("insert into definition
       (definition_id, definition, word_id) " +
     "values (?, ?, ?)" , [definitionId, definition, wordId])
   }
 //has a list of synonyms has been passed in
 if(defsAndSyms.length > 2){
   def synonyms = defsAndSyms[2]
   synonyms.each{ syn ->
     sql.execute("insert into synonym
         (synonym_id, word_id, spelling) values (?, ?, ?)" ,
       [id++, wordId, syn])
     }
   }
}
```

在清单 11 中，主键逻辑过于简单了，不过我可以用几个测试证明我的信心。

清单 12. 创建单词逻辑的测试用例

```
void testCreateWord() {
 dictionary.bloviate("Verb",
      ["To discourse at length in a pompous or boastful manner"],
   ["orate", "gabble", "lecture"])
 def val = dictionary.bloviate()
 def expect = "To discourse at length in a pompous or boastful mar
 assertEquals("should be: " + expect, expect, val[0])
}
void testCreateWordNoSynonyms() {
 dictionary.echelon("Noun", ["a level within an organization"])
 def val = dictionary.echelon()
 def expect = "a level within an organization"
 assertEquals("should be: " + expect, expect, val[0])
}
```

与通常一样，我在清单 12 中编写了几个测试用例以验证它按预想的那样工作。创建单词后，我查询这个单词的 `dictionary` 实例以确认它被加入了数据库。

5. 获取单词的定义

任何传递给字典应用程序的、不匹配前面任何查询类型（词性和同义词）、并且不包含任何参数的消息都被认为是定义查询。例如，一个像 `.glib` 或者 `.glib()` 的消息会返回 *glib*的定义。

清单 **13.** 查单词的定义并不困难！

```
private getDefinitions(word){
 def definitions = []
 sql.eachRow("select definition.definition from definition, word '
   "where definition.word_Id = word.word_id and " +
   "word.spelling = ?", [word]){ arow ->
     definitions << arow.definition
   }
 return definitions
}
```

因为我同时允许让方法调用和属性调用作为定义查询，因此在清单 14 中必须编写至少两个测试。就像在清单 6 中一样，可以假定 *pugnacious*已经在数据库中了。DbUnit 是不是很方便？

清单 **14.** 测试两种情况 —— 方法调用和属性

```
void testFindWord() {
 def val = dictionary.pugnacious()
 def expect = "Combative in nature; belligerent."
 assertEquals("should be: " + expect, expect, val[0])
}
void testFindWordAsProperty() {
 def val = dictionary.pugnacious
 def expect = "Combative in nature; belligerent."
 assertEquals("should be: " + expect, expect, val[0])
}
```

这就是字典应用程序的语义。现在让我们更深入地分析应用程序背后的逻辑。

MOP 逻辑

字典应用程序的关键是 `invokeMethod` 和 `getProperty` 的实现。 我要在这些方法中做出智能的决定，即在向应用程序传递消息之后，应该如何继续。决定是通过一组条件语句而确定的，它们进行 `String` 操作以确定消息的类型。

清单 15 中 惟一麻烦的条件就是第一条，它要验证消息是一个定义查询而不是其他可能的组合，如 `is...` 、 `remove...` 、 `delete...` 或者 `synonymOf...` 。

清单 **15. MOP** 的核心逻辑

```
def invokeMethod(String methodName, Object params) {
 if(isGetDefinitions(methodName, params)){
   return getDefinitions(methodName)
 }else if (params.length > 0){
   createWord(methodName, params)
 }else if(methodName[0..1] == 'is'){
   return determinePartOfSpeech(methodName)
 }else if
 (methodName[0..5] == 'remove' || methodName[0..5] == 'delete'){
      removeWord(methodName)
 }else if (methodName[0..9] == 'synonymsOf'){
   return getSynonyms(methodName)
 }
}

private isGetDefinitions(methodName, params){
 return !(params.length > 0) &&
   ( (methodName.length() <= 5
   && methodName[0..1] != 'is' ) ||
   (methodName.length() <= 10
   && isRemoveDeleteIs(methodName) ) ||
   (methodName.length() >= 10
   && methodName[0..9] != 'synonymsOf'
   && isRemoveDeleteIs(methodName)))
}

private isRemoveDeleteIs(methodName){
    return (methodName[0..5] != 'remove'
     && methodName[0..5] != 'delete'
     && methodName[0..1] != 'is')
 }
```

`isGetDefinitions` 方法做了很多工作，其中一些工作依赖于 `isRemoveDeleteIs` 。当它确定一个消息不符合定义查询时，逻辑就变得简单多了。

清单 16 中的 `getProperty` 方法很简单，因为我规定用户只能按属性查询定义，因此，调用 `getProperty` 时，我调用 `getDefinitions` 方法。

清单 **16.** 太好了，属性很简单！

```
def getProperty(String property){
 return getDefinitions(property)
}
```

就是这样了！真的。清单 5 到 16 中的逻辑创建了一个应用程序，它给予用户相当的灵活性，同时又不会太复杂。当然，如前所述，应用程序中的主键逻辑并不是很保险。在这方面有多种改进方法，从数据库序列到像 Hibernate 这样的框架都可以，Hibernate 框架用于相当得体地处理 ID。

眼见为实，清单 17 展示了字典应用程序的语言的实际能力。（如果在使用 SAT 之前有这样的东西该多好！）

清单 **17.** 字典展示

```
import com.vanward.groovy.SimpleDictionary
def dict = new SimpleDictionary()
dict.vanward("Adjective", ["Being on, or towards the front"],
    ["Advanced"])
dict.pulchritude("Noun", ["Great physical beauty and appeal"])
dict.vanward             //prints "Being on, or towards the front'
dict.pulchritude()       //prints "Great physical beauty and appea
dict.synonymsOfVanward() //prints "Advanced"
dict.isVanwardANoun()    //prints "false"
dict.removeVanward()
dict.deletePulchritude()
```

MOP 的好处

现在，您可能会问 有什么好处？我本可以轻易地公开这些 `private` 方法，重新安排几项内容，就可以给出一个正常工作的应用程序（即，可以公开 `getDefinition` 方法、 `createWord` 方法等），而无需使用 MOC。

让我们分析这种提议 —— 假定我要不使用 Goovy 的 MOC 实现来创建完全相同的字典应用程序。我需要重新定义 `createWord()` 方法的参数清单，使它变成类似 `def createWord(word, partOfSpeech, defs, syns=[]){}` 的样子。

可以看出，第一步是去掉 `private` 修饰符，用 `def` 替换它，这与将方法声明为 `public` 是一样的。然后，要定义参数，并让最后一个参数为可选的（ `syns=[]` ）。

还要定义一个 `delete` 方法，它至少要调用 `remove` 方法以同时支持 remove 和 delete。词性查询也应该修改。可以加入另一个参数，这样它看起来像 `determinePartOfSpeech(word, partOfSpeech)` 这样。

实际上，经过以上步骤，字典应用程序的概念复杂性就从 MOP 实现转移到了静态 API 上了。这就带来了这个问题 —— 这样做的好处是什么？当然您可以看出由于实现 MOP，我为应用程序的使用者提供了无可比拟的灵活性。得到的 API 是一组静态定义的方法，灵活完全比不上 语言本身！

结束语

如果现在头有些发晕了，那么想一下这个：如果可以构建这样一个字典应用程序，使用户可以随意创建查询会怎么样呢？例如，`findAllWordsLikeGlib` 或者 `findAllWordsWithSynonymGlib` 等等，这些语义对于 MOP 来说这只是文字解析罢了，而对用户来说，则是强大的查询功能！

现在再进一步：如果可以创建另一种迷你语言，它对一项业务更有用一些，比如针对股市交易的迷你语言，那会怎么样呢？如果更进一步，将这个应用程序构建为有友好的控制台呢？不用编写脚本，用户使用的是 Groovy 的 shell。（记住我是怎么说 Bash 的？）

如果您认为这种“炼金术”工具只是痴迷于 Emacs 扩展的古怪的 LISP 家伙们用的，那就错了！只要看看马路对面那些喧嚷的 Ruby on Rails 老手们就会知道这种平台是多么的令人激动。深入了解您就会看到当 MOP 发挥作用时，只要 遵守命名规范，不用深入 SQL 就可创建很好的查询。谁在那儿乐呢？

下载

描述	名字	大小
Sample code	j-groovy-mop.tar.gz	5 KB

实战 Groovy: 用 curry 过的闭包进行函数式编程

Groovy 日常的编码构造到达了闭包以前没有到过的地方

在 Groovy 中处处都是闭包，Groovy 闭包惟一的问题是：当每天都使用它们的时候，看起来就有点平淡了。在本文中，客座作者 Ken Barclay 和 John Savage 介绍了如何对标准的闭包（例如闭包复合和 Visitor 设计模式）进行 curry 处理。`curry()` 方法是由 Haskell Curry 发明的，在 JSR 标准发布之前就已经在 Groovy 语言中了。

几乎从一年前实战 *Groovy* 系列开始，我就已经提供了多个让您了解闭包的机会。在首次作为 *alt.lang.jre* 系列的一部分写 Groovy 时（“感受 Groovy”，2004 年 8 月），我介绍了 Groovy 的闭包语法，而且就在上一期文章中，我介绍了最新的 JSR 标准对相同语法的更新。学习至今，您知道了 Groovy 闭包是代码块，可以被引用、带参数、作为方法参数传递、作为返回值从方法调用返回。而且，它们也可以作为其他闭包的参数或返回值。因为闭包是 `Closure` 类型的对象，所以它们也可以作为类的属性或集合的成员。

虽然这些技术都是很神奇的，但是本期文章要学习的闭包技术要比您迄今为止尝试过的都要更火辣一些。客座作者 John Savage 和 Ken Barclay 已经用 Groovy 闭包中的 `curry()` 方法做了一些有趣的实验，我们非常有幸可以见识他们的技艺。

关于本系列

把任何一个工具集成进开发实践的关键是，知道什么时候使用它而什么时候把它留在箱子中。脚本语言可以是工具箱中极为强大的附件，但是只有在恰当地应用到适当的场景中时才是这样。为了这个目标，实战 *Groovy* 是一系列文章，专门介绍 Groovy 的实际应用，并教您何时以及如何成功地应用它们。

Barclay 和 Savage 的 curry 过的闭包不仅会重新激起您对熟悉的操作（例如复合）和 Visitor 设计模式的兴奋，还会用 Groovy 打开函数式编程的大门。

进行 curry 处理

curry 过的函数 一般可在函数式编程语言（例如 ML 和 Haskell）中找得到（请参阅参考资料）。*curry* 这个术语来自 Haskell Curry，这个数学家发明了局部函数的概念。*Currying* 指的是把多个参数放进一个接受许多参数的函数，形成一个新的函数接受余下的参数，并返回结果。令人兴奋的消息是，Groovy 的当前版本（在编写本文时是 jsr-02 版）支持在 `闭包` 上使用 `curry()` 方法 —— 这意味着，我们这些 Groovy 星球的公民，现在可以利用函数式编程的某些方面了！

您以前可能从未用过 `curry()`，所以我们先从一个简单的、熟悉的基础开始。清单 1 显示了一个叫做 `multiply` 的闭包。它有形式参数 `x` 和 `y`，并返回这两个值的乘积。假设没有歧义存在，然后代码演示了用于执行 `multiply` 闭包的两

种方法：显式（通过 `call` 的方式）或隐式。后一种样式引起了函数调用方式。

清单 **1.** 简单的闭包

```
def multiply = { x, y -> return x * y } // closure
def p = multiply.call(3, 4)             // explicit call
def q = multiply(4, 5)                  // implicit call
println "p: ${p}"                       // p is 12
println "q: ${q}"                       // q is 20
```

这个闭包当然很好，但是我们要对它进行 curry 处理。在调用 `curry()` 方法时，不需要提供所有的实际参数。*curry* 过的 调用只引起了闭包的部分应用程序。闭包的 部分应用程序 是另一个 `Closure` 对象，在这个对象中有些值已经被修正。

清单 2 演示了对 `multiply` 闭包进行 curry 处理的过程。在第一个例子中，参数 `x` 的值被设置为 3。名为 `triple` 的闭包现在有效地拥有了 `triple = { y -> return 3 * y }` 的定义。

清单 **2. curry** 过的闭包

```
def multiply = { x, y -> return x * y }  // closure
def triple = multiply.curry(3)           // triple = { y -> return
def quadruple = multiply.curry(4)
// quadruple = { y -> return 4 * y }
def p = triple.call(4)                   // explicit call
def q = quadruple(5)                     // implicit call
println "p: ${p}"                        // p is 12
println "q: ${q}"                        // q is 20
```

可以看到，参数 `x` 已经从 `multiply` 的定义中删除，所有它出现的地方都被 3 这个值代替了。

curry 过的数学 101

从基本数学可能知道，乘法运算符是可交换的（换句话说 `x * y = y * x` ）。但是，减法运算符是不可交换的；所以，需要两个操作来处理减数和被减数。清单 3 为这个目的定义了闭包 `lSubtract` 和 `rSubtract` （分别在左边和右边），结果显示了 `curry` 函数的一个有趣的应用。

清单 **3.** 右和右操作数

```
def lSubtract = { x, y -> return x - y }
def rSubtract = { y, x -> return x - y }
def dec = rSubtract.curry(1)
 // dec = { x -> return x - 1 }
def cent = lSubtract.curry(100)
// cent = { y -> return 100 - y }
def p = dec.call(5)                         // explicit call
def q = cent(25)                            // implicit call
println "p: ${p}"                           // p is 4
println "q: ${q}"                           // q is 75
```

迭代和复合

您会回忆起这个系列以前的文章中，闭包一般用于在 `List` 和 `Map` 集合上应用的迭代器方法 上。例如，迭代器方法 `collect` 把闭包应用到集合中的每个元素上，并返回一个带有新值的新集合。清单 4 演示了把 `collect` 方法应用于 `List` 和 `Map`。名为 `ages` 的 `List` 被发送给 `collect()` 方法，使用单个闭包 `{ element -> return element + 1 }` 作为参数。注意方法的最后一个参数是个闭包，在这个地方 Groovy 允许把它从实际参数列表中删除，并把它直接放在结束括号后面。在没有实际参数时，可以省略括号。用名为 `accounts` 的 `Map` 对象调用的 `collect()` 方法可以展示这一点。

清单 **4.** 闭包和集合

```
def ages = [20, 30, 40]
def accounts = ['ABC123' : 200, 'DEF456' : 300, 'GHI789' : 400]
def ages1 = ages.collect({ element -> return element + 1 })
def accounts1 = accounts.collect
  { entry -> entry.value += 10; return entry }
println "ages1: ${ages1}"
// ages1: [21, 31, 41]
println "accounts1: ${accounts1}"
// accounts1: [ABC123=210, GHI789=410, DEF456=310]
def ages2 = ages.collect { element -> return dec(element) }
println "ages2: ${ages2}"
 // ages2: [19, 29, 39]
```

最后一个例子搜集名为 `ages` 的 `List` 中的所有元素，并将 `dec` 闭包（来自清单 3）应用于这些元素。

闭包复合

闭包更重要的一个特征可能就是复合（*composition*），在复合中可以定义一个闭包，它的目的是组合其他闭包。使用复合，两个或多个简单的闭包可以组合起来构成一个更复杂的闭包。

清单 5 介绍了一个漂亮的 `composition` 闭包。现在请注意仔细阅读：参数 `f` 和 `g` 代表 单个参数闭包。到现在还不错？现在，对于某些参数值 `x`，闭包 `g` 被应用于 x，而闭包 `f` 被应用于生成的结果！哦，只对前两个闭包参数进行了 curry 处理，就有效地形成了一个组合了这两个闭包的效果的新闭包。

清单 5 组合了闭包 `triple` 和 `quadruple`，形成闭包 `twelveTimes`。当把这个闭包应用于实际参数 3 时，返回值是 36。

清单 **5.** 超级闭包复合

```
def multiply = { x, y -> return x * y }
// closure
def triple = multiply.curry(3)
// triple = { y -> return 3 * y }
def quadruple = multiply.curry(4)
// quadruple = { y -> return 4 * y }
def composition = { f, g, x -> return f(g(x)) }
def twelveTimes = composition.curry(triple, quadruple)
def threeDozen = twelveTimes(3)
println "threeDozen: ${threeDozen}"
// threeDozen: 36
```

非常漂亮，是么？

五星计算

现在我们来研究闭包的一些更刺激的方面。我们先从一个机制开始，用这个机制可以表示包含计算模式 的闭包，计算模式是一个来自函数式编程的概念。计算模式的一个例子就是用某种方式把 `List` 中的每个元素进行转换。因为这些模式发生得如此频繁，所以我们开发了一个叫做 `Functor` 的类，把它们封装成 `static Closure`。清单 6 显示了一个摘要。

清单 **6. Functor** 封装了一个计算模式

```
package fp
abstract class Functor {
 //  arithmetic (binary, left commute and right commute)
 public static Closure bMultiply     = { x, y -> return x * y }
 public static Closure rMultiply     = { y, x -> return x * y }
 public static Closure lMultiply     = { x, y -> return x * y }

 // ...
 // composition
 public static Closure composition   = { f, g, x -> return f(g(x))

 // lists
 public static Closure map    =
   { action, list -> return list.collect(action) }

 public static Closure apply  = { action, list -> list.each(action)

 public static Closure forAll = { predicate, list ->
                                  for(element in list) {
                                    if(predicate(element) == false)
                                        return false
                                    }
                                  }
                                  return true
                               }
    // ...
}
```

在这里可以看到名为 `map` 的闭包，不要把它与 `Map` 接口混淆。`map` 闭包有一个参数 `f` 代表闭包，还有一个参数 `list` 代表（不要惊讶） `List` 。它返回一个新 `List` ，其中 `f` 已经映射到 `list` 中的每个元素。当然，Groovy 已经有了用于 `Lists` 的 `collect()` 方法，所以我们在我们的实现中也使用了它。

在清单 7 中，我们把事情又向前进行了一步，对 `map` 闭包进行了 *curry* 处理，形成一个块，会将指定列表中的所有元素都乘以 12。

清单 **7.** 添加一些 **curry**，并乘以 **12**

```
import fp.*
def twelveTimes = { x -> return 12 * x }
def twelveTimesAll = Functor.map.curry(twelveTimes)
def table = twelveTimesAll([1, 2, 3, 4])
println "table: ${table}"
// table: [12, 24, 36, 48]
```

现在，这 就是我们称之为五星计算的东西！

业务规则用 curry 处理

关于闭包的技艺的讨论很不错，但是更关注业务的人会更欣赏下面这个例子。在考虑计算特定 `Book` 条目净值的问题时，请考虑商店的折扣和政府的税金（例如 增值税）。如果想把这个逻辑作为 `Book` 类的一部分包含进来，那么形成的解决方案可能是个难缠的方案。因为书店可能会改变折扣，或者折扣只适用于选定的存货，所以这样一个解决方案可能会太刚性了。

但是猜猜情况如何。变化的业务规则非常适合于使用 curry 过的闭包。可以用一组简单的闭包来表示单独的业务规则，然后用复合把它们以不同的方式组合起来。最后，可以用 计算模式 把它们映射到集合。

清单 8 演示了书店的例子。闭包 `rMultiply` 是个局部应用程序，通过使用一个不变的第二操作数，把二元乘法改变成一元闭包。两个图书闭包 `calcDiscountedPrice` 和 `calcTax` 是 `rMultiply` 闭包的实例，为乘数值设置了值。闭包 `calcNetPrice` 是计算净价的算法：先计算折扣价格，然后在折扣价格上计算销售税。最后， `calcNetPrice` 被应用于图书价格。

清单 **8.** 图书业务对象

```
import fp.*
class Book {
    @Property name
    @Property author
    @Property price
    @Property category
}
def bk = new Book(name : 'Groovy', author :
  'KenB', price : 25, category : 'CompSci')
// constants
def discountRate = 0.1
def taxRate = 0.17
//  book closures
def calcDiscountedPrice = Functor.rMultiply.curry(1 - discountRate)
def calcTax = Functor.rMultiply.curry(1 + taxRate)
def calcNetPrice =
  Functor.composition.curry(calcTax, calcDiscountedPrice)
//  now calculate net prices
def netPrice = calcNetPrice(bk.price)
println "netPrice: ${netPrice}"        // netPrice: 26.325
```

更加 groovy 的访问者

已经看到了如何把 curry 过的闭包应用于函数模式，所以现在我们来看看在使用相似的技术重新访问重要的面向对象设计模式时发生了什么。对于面向对象系统来说，必须遍历对象集合并在集合中的每个元素上执行某个操作，是非常普通的使用情况。假设一个不同的场景，系统要遍历相同的集合，但是要执行不同的操作。通常，需要用 *Visitor* 设计模式（请参阅 参考资料）来满足这一需求。 `Visitor` 接口引入了处理集合元素的动作协议。具体的子类定义需要的不同行为。方法被引进来遍历集合并对每个元素应用 `Visitor` 动作。

如果到现在您还没猜出来，那么可以用闭包实现同的效果。这种方法的一个抽象是：使用闭包，不需要开发访问者类的层次结构。而且，可以有效地使用闭包复合和映射来定义集合的动作和效果遍历。

例如，考虑用来表示图书馆库存的类 `Library` 和类 `Book` 之间的一对多关系。可以用 `List` 或 `Map` 实现这个关系；但是 `Map` 提供的优势是它能提供快速的查询，也就是说用图书目录编号作为键。

清单 9 显示了一个使用 `Map` 的简单的一对多关系。请注意 `Library` 类中的两个显示方法。引入访问者时，两者都是重构的目标。

清单 **9.** 图书馆应用程序

```
class Book {
    @Property title
    @Property author
    @Property catalogNumber
    @Property onLoan = false
    String toString() {
        return "Title: ${title}; author: ${author}"
    }
}
class Library {
    @Property name
    @Property stock = [ : ]

    def addBook(title, author, catalogNumber) {
        def bk = new Book(title : title, author :
          author, catalogNumber : catalogNumber)
        stock[catalogNumber] = bk
    }

    def lendBook(catalogNumber) {
        stock[catalogNumber].onLoan = true
    }

    def displayBooksOnLoan() {
        println "Library: ${name}"
        println "Books on loan"
        stock.each { entry ->
            if(entry.value.onLoan == true) println entry.value
        }
    }

    def displayBooksAvailableForLoan() {
        println "Library: ${name}"
        println "Books available for loan"
        stock.each { entry ->
            if(entry.value.onLoan == false) println entry.value
        }
    }
}
def lib = new Library(name : 'Napier')
lib.addBook('Groovy', 'KenB',
  'CS123')
lib.addBook('Java', 'JohnS', 'CS456')
lib.addBook('UML', 'Ken and John',
  'CS789')
lib.lendBook('CS123')
lib.displayBooksOnLoan()    // Title: Groovy; author: KenB
lib.displayBooksAvailableForLoan()    // Title: UML; author: Ken ar
    // Title: Java; author: JohnS
```

清单 10 包含 `Library` 类中的几个闭包，模拟了访问者的用法。`action` 闭包（与 `map` 闭包有点相似）把 `action` 闭包应用于 `List` 的每个元素。如果某本书被借出，则闭包 `displayLoanedBook` 显示它；如果某本书未被借出，则闭包 `displayAvailableBook` 显示它。两者都扮演访问者和相关的动作。用 `displayLoanedBook` 对 `apply` 闭包进行 curry 处理，会形成闭包 `displayLoanedBooks`，它为处理图书集合做好了准备。类似的方案也用来生成可供借阅的图书显示，如清单 10 所示。

清单 10. 修订后的图书馆访问者

```
import fp.*
class Book {
    @Property title
    @Property author
    @Property catalogNumber
    @Property onLoan = false
    String toString() {
        return "    Title: ${title}; author: ${author}"
    }

}
class Library {
    @Property name
    @Property stock = [ : ]
    def addBook(title, author, catalogNumber) {
        def bk = new Book(title : title, author :
          author, catalogNumber : catalogNumber)
        stock[catalogNumber] = bk
    }

    def lendBook(catalogNumber) {
        stock[catalogNumber].onLoan = true
    }

    def displayBooksOnLoan() {
        println "Library: ${name}"
        println "Books on loan"
        displayLoanedBooks(stock.values())
    }

    def displayBooksAvailableForLoan() {
        println "Library: ${name}"
        println "Books available for loan"
        displayAvailableBooks(stock.values())
    }

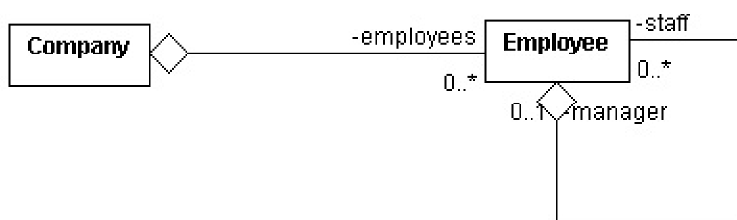
    private displayLoanedBook = { bk -> if(bk.onLoan == true)
      println bk }
    private displayAvailableBook = { bk -> if(bk.onLoan == false)
      println bk }
```

```
    private displayLoanedBooks =
      Functor.apply.curry(displayLoanedBook)
    private displayAvailableBooks =
      Functor.apply.curry(displayAvailableBook)
}
def lib = new Library(name : 'Napier')
lib.addBook('Groovy', 'KenB',
  'CS123')
lib.addBook('Java', 'JohnS', 'CS456')
lib.addBook('UML', 'Ken and John',
  'CS789')
lib.lendBook('CS123')
lib.displayBooksOnLoan()    // Title: Groovy; author: KenB
lib.displayBooksAvailableForLoan() // Title: UML; author: Ken and J
                                   // Title: Java; author: JohnS
```

用闭包进行测试

在结束之前，我们来看一下 Groovy 闭包的一个附加用途。请考虑一个被建模为具有许多 `Employee` 的 `Company` 的应用程序。递归的关系在单个 `Employee`（团队领导）和许多 `Employee`（团队成员）之间进一步建立起一对多的聚合。图 1 是这样一个组织的类图。

图 1. Company 应用程序



可以用闭包来表述模型架构上的完整性。例如，在这个例子中，可能想确保每个员工都分配了一个经理。简单的闭包 `hasManager` 为每个员工表达了这个需求：`def hasManager = { employee -> return (employee.manager != null)` 。

来自清单 6 的 `Functor` 类中的 `forAll` 闭包的局部应用程序能够描述架构的需求：`def everyEmployeeHasManager = Functor.forAll.curry(hasManager)` 。

清单 11 演示了 curry 过的闭包的应用：测试系统架构的完整性。

清单 11. 用于测试架构完整性的闭包

```
import fp.*
/**
 *  A company with any number of employees.
 *  Each employee is responsible
 *  to a team leader who, in turn, manages a team of staff.
 */
import java.util.*
class Employee {
    @Property id
    @Property name
    @Property staff = [ : ]
    @Property manager = null
    String toString() {
        return "Employee: ${id} ${name}"
    }

    def addToTeam(employee) {
        staff[employee.id] = employee
        employee.manager = this
    }

}
class Company {
    @Property name
    @Property employees = [ : ]
    def hireEmployee(employee) {
        employees[employee.id] = employee
    }

    def displayStaff() {
        println "Company: ${name}"
        println "===================="
        employees.each { entry -> println "
          ${entry.value}" }
    }

}
def co = new Company(name : 'Napier')
def emp1 = new Employee(id : 123, name : 'KenB')
def emp2 = new Employee(id : 456, name : 'JohnS')
def emp3 = new Employee(id : 789, name : 'JonK')
co.hireEmployee(emp1)
co.hireEmployee(emp2)
co.hireEmployee(emp3)
emp3.addToTeam(emp1)
emp3.addToTeam(emp2)
co.displayStaff()
//  Architectural closures
def hasManager = { employee -> return (employee.manager != null) }
def everyEmployeeHasManager = Functor.forAll.curry(hasManager)
def staff = new ArrayList(co.employees.values())
println "Every employee has a manager?:
```

```
    ${everyEmployeeHasManager.call(staff)}"    // false
```

curry 是优秀的

在本文中您已看到了大量闭包，但是希望能激起您对更多闭包的渴望。在学习乘法例子时，curry 过的闭包使得实现计算的函数模式出奇得容易。一旦掌握了这些模式，就可以把它们部署到常见的企业场景中，例如我们在书店例子中把它们应用于业务规则。把闭包应用于函数模式是令人兴奋的，一旦这么做了之后，再把它们应用于面向对象设计模式，就不是什么大事情了。Curry 过的闭包可以用来模拟 Visitor 模式的基本元素，正如在 `Library` 例子中显示的。它们在软件测试期间执行完整性测试时也会有用，就像用 `Company` 例子所展示的。

本文中看到的全部例子都是企业系统常见的用例。看到 Groovy 闭包和 `curry` 方法能够如此流畅地应用于众多编程场景、函数模式和面向对象模式，真是激动人心。Haskell Curry 肯定发现了这个可怕的 *groovy*！

实战 Groovy: Groovy 的腾飞

熟悉 *Groovy* 新的、遵循 *JSR* 的语法

随着 Groovy JSR-1（及其后续发行版本）的发布，Groovy 语法的变化已经规范化 —— 这意味着如果以前没有对此加以注意，那么现在是开始注意它的时候了。这个月，Groovy 的常驻实践者 Andrew Glover 将介绍 Groovy 语法最重要的变化，以及在经典 Groovy 中找不到的一个方便特性。

从我在 *alt.lang.jre*的系列文章“Feeling Groovy”中介绍 Groovy 开始，差不多有一年时间了。从那时起，通过发行许多版本，逐步解决了语言实现中的问题，并满足开发人员社区的一些特性请求，Groovy 已经成熟了许多。最后，在今年四月，Groovy 有了一个重大飞跃，它正式发布了新的解析器，该解析器的目标就是将这门语言标准化为 JSR 进程的一部分。

在本月“实战 *Groovy*”这一期的文章中，我将祝贺 Groovy 的成长，介绍通过 Groovy 非常好用的新解析器规范化的一些最重要的变化：即变量声明和闭包。因为我要将一些新 Groovy 语法与我在关于 Groovy 的第一篇文章中介绍的经典语法进行比较，所以您可以在另一个浏览器窗口中打开“alt.lang.jre: 感受 Groovy”这篇文章。

为什么会发生这些变化？

如果您一直在跟踪 Groovy，不管是阅读文章和 blog，还是自己编写代码，您都可能已经遭遇过这门语言的一、两个微妙的问题。在进行一些灵敏的操作，例如对象导航，特别是使用闭包的时候，Groovy 偶尔会遇到歧义性问题和语法受限的问题。几个月之前，作为 JSR 进程的一部分，Groovy 团队开始着手解决这些问题。四月份，随 groovy-1.0-jsr-01 发行版本提供的解决方案是一个更新过的语法以及一个用来对语法进行标准化的新语法内容解析器。

关于本系列

将任何一个工具集成到开发实践中的关键是：知道什么时候使用它，什么时候把它留在箱子中。脚本语言可以是工具箱中极为强大的附件，但是只有在将它恰当应用到适当的地方时才这样。为了实现这个目标，实战 *Groovy*是一个文章系列，专门介绍 Groovy 的实际应用，并教导您什么时候使用它们，以及如何成功地应用它们。

好消息是：新语法是对语言的完全增强。另一个好消息是：它和以前的语法没有很大不同。像所有 Groovy 一样，语法的设计目标是较短的学习曲线和较大的回报。

当然，符合 JSR 的解析器也给新 Groovy 带来一些与目前“经典”语法不兼容的地方。如果用新的解析器运行本系列以前文章中的代码示例，那么您自己就可以看，代码示例可能无法工作！现在，这一点看起来可能有点苛刻 —— 特别是对 Groovy

这样自由的语言来说 —— 但是解析器的目标是确保作为 Java 平台的 标准化语言的 Groovy 的持续成长。可以把它当作新 Groovy 的一个有帮助的指南。

Groovy 依然是 Groovy ！

在深入研究变化的内容之前，我要花几秒钟谈谈什么 没有改变。首先，动态类型化的基本性质没有改变。变量的显式类型化（即将变量声明为 `String` 或 `Collection` ）依然是可选的。稍后，我会讨论对这一规则的一点新增内容。

知道分号依然是可选的时候，许多人都会感到轻松。对于放松对这个语法的使用，存在许多争论，但是最终少数派赢得了胜利。底线是：如果愿意，也可以使用分号。

集合（Collection）的使用大部分还保持不变。仍然可以用 *array*语法和 `map` ，像以前那样（即最初从文章“alt.lang.jre: 感受 Groovy”中学到的方式）声明类似 `list` 的集合。但范围上略有变化，我将在后面部分展示这一点。

最后，Groovy 对标准 JDK 类的增加没有发生多少变化。语法糖衣和漂亮的 API 也没变， 就像普通的 Java `File` 类型的情况一样，我稍后将展示这一点。

容易变的变量

Groovy 的变量规则对新的符合 JSR 的语法的打击可能最大。经典的 Groovy 在变量声明上相当灵活（而且实际上很简洁）。而使用新的 JSR Groovy 时，所有的变量前都必须加上 `def` 关键字或者 `private` 、 `protected` 或 `public` 这样的修饰符。当然，也可以声明变量类型。另外，如果正在定义类，希望声明属性（使用 JavaBeans 样式的 getter 和 setter 公开），那么也可以用 `@Property` 关键字声明成员变量。请注意 —— *Property*中的 *P*是大写的！

例如，在“alt.lang.jre: 感受 Groovy”中介绍 *GroovyBeans*时，我在文章的清单 22 中定义了一个叫做 `LavaLamp` 的类型。这个类不再符合 JSR 规范，如果要运行它，则会造成解析器错误。幸运的是，迁移这个类不是很困难：我要做的全部工作就是给所有需要的成员变量添加 `@Property` 属性，如清单 1 所示：

清单 **1. LavaLamp** 的返回结果

```
package com.vanward.groovy
class LavaLamp{
 @Property model
 @Property baseColor
 @Property liquidColor
 @Property lavaColor
}
llamp = new LavaLamp(model:1341, baseColor:"Black",
 liquidColor:"Clear", lavaColor:"Green")
println llamp.baseColor
println "Lava Lamp model ${llamp.model}"
myLamp = new LavaLamp()
myLamp.baseColor = "Silver"
myLamp.setLavaColor("Red")
println "My Lamp has a ${myLamp.baseColor} base"
println "My Lava is " + myLamp.getLavaColor()
```

不是太坏，不是吗？

正如上面描述的，对于任何变量，如果没有修饰符、`@Property` 关键字或者类型，则需要具有 `def` 关键字。例如，清单 2 的代码在 `toString` 方法中包含一个中间变量 `numstr` ，如果用 JSR 解析器运行此代码，则会造成一个错误：

清单 **2.** 不要忘记 **def** 关键字！

```
class Person {
 @Property fname
 @Property lname
 @Property age
 @Property address
 @Property contactNumbers
 String toString(){

  numstr = new StringBuffer()

  if (contactNumbers != null){
      contactNumbers.each{
        numstr.append(it)
        numstr.append(" ")
      }
  }

  "first name: " + fname + " last name: " + lname +
   " age: " + age + " address: " + address +
   " contact numbers: " + numstr.toString()
}
}
```

认识这个代码吗？它借用来自“在 Java 应用程序中加一些 Groovy 进来”一文的清单 1 中的代码。在清单 3 中，可以看到运行代码时弹出的错误消息：

清单 **3.** 错误消息

```
c:\dev\projects>groovy BusinessObjects.groovy

BusinessObjects.groovy: 13: The variable numstr is undefined in th
@ line 13, column 4\.
     numstr = new StringBuffer()
     ^
1 Error
```

当然，解决方案也是在 `toString` 方法中将 `def` 关键字添加到 `numstr` 。清单 4 显示了这个更合适的 *def*解决方案。

清单 **4.** 用 **def** 重新处理

```
 String toString(){

  def numstr = new StringBuffer()

  if (contactNumbers != null){
      contactNumbers.each{
        numstr.append(it)
        numstr.append(" ")
      }
  }

  "first name: " + fname + " last name: " + lname +
   " age: " + age + " address: " + address +
   " contact numbers: " + numstr.toString()
}
```

另外，我还可以为 `numstr` 提供一个像 `private` 这样的修饰符，或者将它声明为 `StringBuffer` 。不论哪种方法，重要的一点是：在 JSR Groovy 中，必须在变量前加上 某些东西。

封闭闭包（**closure**）

闭包的语法发生了变化，但是大多数变化只与参数有关。在经典的 Groovy 中，如果为闭包声明参数，就必须用 `|` 字符作为分隔符。您可能知道， `|` 也是普通 Java 语言中的位操作符；结果，在 Groovy 中，只有在某个闭包的参数声明的上下文中，才能使用 `|` 字符。

在“alt.lang.jre: 感受 Groovy”的清单 21 中，我演示了迭代，查看了用于闭包的经典 Groovy 参数语法。可以回想一下，我在集合上运用了 `find` 方法，该方法试图找到值 *3*。然后我传入了参数 `x` ，它代表 `iterator` 的下一个值（有经验的 *Groovy*开发人员会注意到， `x` 完全是可选的，我可以引用隐式变量 `it` ）。使用 JSR Groovy 时，必须删除 `|` ，并用 Nice 样式的 `->` 分隔符代替它，如清单 5 所示：

清单 **5.** 新的 **Groovy** 闭包语法

```
[2, 4, 6, 8, 3].find { x ->
 if (x == 3){
   println "found ${x}"
 }
}
```

新的闭包语法有没有让您想起 Nice 语言的块语法呢？如果不熟悉 Nice 语言，请参阅 alt.lang.jre: Nice 的双倍功能，这是我在 *alt.lang.jre*系列上贡献的另一篇文章。

正如我在前面提到过的，Groovy 的 JDK 没有变。但是就像刚才所学到的，闭包却发生了变化；所以，使用 Groovy 的 JDK 中那些漂亮的 *API*的方式也发生了变化，但仅仅是轻微的变化。在清单 6 中，可以看到这些变化对 Groovy IO 的影响根本是微不足道的：

清单 **6. Groovy** 的 **JDK** 依旧功能强大！

```
import java.io.File
new File("maven.xml").eachLine{ line ->
 println "read the following line -> " + line
}
```

改编过滤器

现在，不得不让您跳过很大一部分，但您还记得在“用 Groovy 进行 Ant 脚本编程”一文中，我是如何介绍闭包的威力和工具的吗？谢天谢地，我在这个专栏的示例中所做的多数工作都很容易针对新语法重新进行改编。在清单 7 中，我只是将 `@Property` 属性添加到 `Filter` 的成员 `strategy` （最初在那篇文章的清单 2 和清单 3 中显示）。然后在闭包中添加 `->` 分隔符，万岁—— 它可以工作了！

清单 **7.** 过滤改编！

```
package com.vanward.groovy
class Filter{
@Property strategy
boolean apply(str){
 return strategy.call(str)
}
}
simplefilter = { str ->
 if(str.indexOf("java.") >= 0){
   return true
 }else{
   return false
 }
}

fltr = new Filter(strategy:simplefilter)
assert !fltr.apply("test")
assert fltr.apply("java.lang.String")

rfilter = { istr ->
 if(istr =~ "com.vanward.*"){
   return true
 }else{
   return false
 }
}

rfltr = new Filter(strategy:rfilter)
assert !rfltr.apply("java.lang.String")
assert rfltr.apply("com.vanward.sedona.package")
```

目前为止还不坏，您觉得呢？新的 Groovy 语法很容易掌握！

对范围（range）的更改

Groovy 的范围语法的变化非常小。在经典的 Groovy 中，您可以通过使用 `...` 语法指明排他性（即上界）来避开这些变化。在 JSR Groovy 中，只要去掉最后一个点（ `.` ），并用直观的 `<` 标识替代它即可。

请注意观察我在下面的清单 8 中对来自“Feeling Groovy”一文中的范围示例进行的改编：

清单 **8.** 新的范围语法

```
myRange = 29..<32
myInclusiveRange = 2..5
println myRange.size() // still prints 3
println myRange[0]   // still prints 29
println myRange.contains(32) //  still prints false
println myInclusiveRange.contains(5) // still prints true
```

您是说存在歧义？

您可能注意到，在使用 Groovy 时，有一项微妙的功能可以让您获得方法引用，并随意调用这个引用。可以将方法指针当作调用对象方法的方便机制。关于方法指针，有意思的事情是：它们的使用可能就表明代码违反了迪米特法则。

您可能会问“什么是迪米特法则”呢？迪米特法则使用 只与直接朋友对话这句格言指出：我们应当避免调用由另一个对象方法返回的对象上的方法。例如，如果 `Foo` 对象公开了一个 `Bar` 对象类型，那么客户应当 通过 *`Foo`* 访问 `Bar` 的行为。结果可能是一些脆弱的代码，因为对某个对象的更改会传播到整个范围。

一位受尊敬的学者写了一篇优秀的文章，叫做“The Paperboy, the Wallet, and the Law of Demeter”（请参阅 参考资料）。这篇文章中的示例是用 Java 语言编写的；但是，我在下面用 Groovy 重新定义了这些示例。在清单 9 中，可以看到这些代码演示了迪米特法则 —— 如何用它洗劫人们的钱包！

清单 **9.** 迪米特在行动（同情啊，同情！）

```
package com.vanward.groovy
import java.math.BigDecimal
class Customer {
@Property firstName
@Property lastName
@Property wallet
}
class Wallet {
@Property value;
def getTotalMoney() {
 return value;
}

def setTotalMoney(newValue) {
 value = newValue;
}
def addMoney(deposit) {
 value = value.add(deposit)
}
def subtractMoney(debit) {
 value = value.subtract(debit)
}
}
```

在清单 9 中，有两个定义的类型 —— 一个 `Customer` 和一个 `Wallet` 。请注意 `Customer` 类型公开自己的 `wallet` 实例的方式。正如前面所说的，代码简单的公开方式存在问题。例如，如果我（像原文章的作者所做的那样）添加了一个坏报童，去抢劫那些不知情客户的钱包，又会怎么样？在清单 10 中，我使用了 Groovy 的方法指针来做这件坏事。请注意我是如何使用 Groovy 新的 `&` 方法指针语法，通过 `Customer` 实例夺取对 `subtractMoney` 方法的引用。

清单 10. 添加坏报童 ...

```
iwallet = new Wallet(value:new BigDecimal(32))
victim = new Customer(firstName:"Lane", lastName:"Meyer", wallet:iw
//Didn't *ask* for a dime. Two Dollars.
victim.getWallet().subtractMoney(new BigDecimal("0.10"))
//paperboy turns evil by snatching a reference to the subtractMoney
mymoney = victim.wallet.&subtractMoney
mymoney(new BigDecimal(2)) // "I want my 2 dollars!"
mymoney(new BigDecimal(25)) // "late fees!"
```

现在，不要误会我：方法指针不是为了侵入代码或者获得对人们现金的引用！方法指针只是一种方便的机制。方法指针也很适合于重新连接上您喜爱的 80 年代的老电影！但是，如果您把这些可爱的、漂亮的东西弄湿了，那么它们可就帮不上您的忙了。严格地说，可以把 Groovy 的 `println` 快捷方式当作 `System.out.println` 的隐式方法指针。

如果一直都很留心，那么您可能已经注意到，JSR Groovy 要求我使用新的 `&` 语法来创建 `subtractMoney` 方法的指针。您可能已经猜到，这个添加消除了经典 Groovy 中的歧义性。

一些新东西！

如果在 Groovy 的 JSR 发行版中没有什么 新东西，那就没有意思了，不是吗？谢天谢地，JSR Groovy 引入了 `as` 关键字，它是一个方便的类型转换机制。这个特性与新的对象创建语法关系密切，新的语法可以用类似数组的语法很容易地在 Groovy 中创建 非定制类。所谓非定制，指的是在 JDK 中可以找到的类，例如 `Color` 、 `Point` 、 `File` ，等等。

在清单 11 中，我用这个新语法创建了一些简单类型：

清单 **11. Groovy** 中的新语法

```
def nfile = ["c:/dev", "newfile.txt"] as File
def val = ["http", "www.vanwardtechnologies.com", "/"] as URL
def ival = ["89.90"] as BigDecimal
println ival as Float
```

注意，我用便捷语法创建了一个新 `File` 和 `URL` ，还有 `BigDecimal` ，还要注意的是，我可以用 `as` 把 `BigDecimal` 类型转换成 `Float` 类型。

接下来是什么呢？

JSR 对 Groovy 的规范化过程并没有结束，特别是在有些东西在 Groovy 的当前版本中（在本文发布时是 JSR-2）仍然不起作用的情况下。例如，在新的 Groovy 中，不能用 `do` / `while` 循环。此外，新的 Groovy 还无法完全支持 Java 5.0 的 `for` 循环概念。结果是，可以使用 `in` 语法，但是不能使用新推出的 `:` 语法。

这些都是重要的特性，不能没有，但是不用担心 —— Groovy 小组正在努力工作，争取在未来几个月内实现它们。请参阅 参考资料，下载最新的发行版本，并学习更多关于 JSR Groovy 进程的内容；还请继续关注下个月的“实战 *Groovy*”，下个月我（和两个客座专栏作家）将深入讨论 Groovy 闭包的更精彩的细节。

实战 Groovy: 在 Java 应用程序中加一些 Groovy 进来

嵌入简单的、易于编写的脚本，从而利用 *Groovy* 的简单性

您有没有想过在自己相对复杂的 Java 程序中嵌入 Groovy 简单的、易于编写的脚本呢？在这一期 实战 *Groovy* 系列文章中，Andrew Glover 将介绍把 Groovy 集成到 Java 代码中的多种方法，并解释在什么地方、什么时候适合这么做。

如果您一直在阅读这个系列，那么您应该已经看到有各种各样使用 Groovy 的有趣方式，Groovy 的主要优势之一就是它的生产力。Groovy 代码通常要比 Java 代码更容易编写，而且编写起来也更快，这使得它有足够的资格成为开发工作包中的一个附件。在另一方面，正如我在这个系列中反复强调的那样，Groovy 并不是 —— 而且也不打算成为 —— Java 语言的替代。所以，这里存在的问题是，能否把 Groovy 集成到 Java 的编程实践中？或者说这样做有什么用？什么时候 这样做有用？

这个月，我将尝试回答这个问题。我将从一些熟悉的事物开始，即从如何将 Groovy 脚本编译成与 Java 兼容的类文件开始，然后进一步仔细研究 Groovy 的编译工具（ `groovyc` ）是如何让这个奇迹实现的。了解 Groovy 在幕后做了什么是在 Java 代码中使用 Groovy 的第一步。

注意，本月示例中演示的一些编程技术是 `Groovlets` 框架和 Groovy 的 `GroovyTestCase` 的核心，这些技术我在前面的文章中已经讨论过。

关于本系列

把任何工具集成到自己的开发实践的关键就是知道什么时候使用它，而什么时候应当把它留在箱子里。脚本语言能够成为工具箱中极为强大的附件，但只在将它恰当应用到合适场景时才这样。为此，实战 *Groovy* 的一系列文章专门探索了 Groovy 的实际应用，并告诉您什么时候应用它们，以及如何成功地应用它们。

天作之合？

在本系列中以前的文章中，当我介绍如何 用 Groovy 测试普通 Java 程序 的时候，您可能已经注意到一些奇怪的事：我 编译了 那些 Groovy 脚本。实际上，我将 groovy 单元测试编译成普通的 Java .class 文件，然后把它们作为 Maven 构建的一部分来运行。

这种编译是通过调用 `groovyc` 命令进行的，该命令将 Groovy 脚本编译成普通的 Java 兼容的 .class 文件。例如，如果脚本声明了一个类，那么调用 `groovyc` 会生成至少三个 .class 。文件本身会遵守标准的 Java 规则：.class 文件名称要和声明的类名匹配。

作为示例，请参见清单 1，它创建了一个简单的脚本，脚本声明了几个类。然后，您自己就可以看出 `groovyc` 命令生成的结果：

清单 **1. Groovy** 中的类声明和编译

```
package com.vanward.groovy
class Person {
  fname
  lname
  age
  address
  contactNumbers
  String toString(){

    numstr = new StringBuffer()
    if (contactNumbers != null){
     contactNumbers.each{
       numstr.append(it)
       numstr.append(" ")
     }
    }
    "first name: " + fname + " last name: " + lname +
    " age: " + age + " address: " + address +
    " contact numbers: " + numstr.toString()
  }
}
class Address {
  street1
  street2
  city
  state
  zip
  String toString(){
   "street1: " + street1 + " street2: " + street2 +
     " city: " + city + " state: " + state + " zip: " + zip
  }
}
class ContactNumber {
  type
  number
  String toString(){
   "Type: " + type + " number: " + number
  }
}
nums = [new ContactNumber(type:"cell", number:"555.555.9999"),
    new ContactNumber(type:"office", number:"555.555.5598")]
addr = new Address(street1:"89 Main St.", street2:"Apt #2",
    city:"Utopia", state:"VA", zip:"34254")
pers = new Person(fname:"Mollie", lname:"Smith", age:34,
    address:addr, contactNumbers:nums)
println pers.toString()
```

在清单 1 中，我声明了三个类 —— `Person` 、 `Address` 和 `ContactNumber` 。之后的代码根据这些新定义的类型创建对象，然后调用 `toString()` 方法。迄今为止，Groovy 中的代码还非常简单，但现在来看一下清单 2 中 `groovyc` 产生什么样的结果：

清单 **2. groovyc** 命令生成的类

```
aglover@12d21 /cygdrive/c/dev/project/target/classes/com/vanward/gı
$ ls -ls
total 15
 4 -rwxrwxrwx+ 1 aglover  user   3317 May  3 21:12 Address.class
 3 -rwxrwxrwx+ 1 aglover  user   3061 May  3 21:12 BusinessObjects.
 3 -rwxrwxrwx+ 1 aglover  user   2815 May  3 21:12 ContactNumber.cl
 1 -rwxrwxrwx+ 1 aglover  user   1003 May  3 21:12
   Person$_toString_closure1.class
 4 -rwxrwxrwx+ 1 aglover  user   4055 May  3 21:12 Person.class
```

哇！五个 .class 文件！我们了解 `Person` 、 `Address` 和 `ContactNumber` 文件的意义，但是其他两个文件有什么作用呢？

研究发现， `Person$_toString_closure1.class` 是 `Person` 类的 `toString()` 方法中发现的闭包的结果。它是 `Person` 的一个内部类，但是 `BusinessObjects.class` 文件是怎么回事 —— 它可能是什么呢？

对 清单 1 的深入观察指出：我在脚本主体中编写的代码（声明完三个类之后的代码）变成一个 .class 文件，它的名称采用的是脚本名称。在这个例子中，脚本被命名为 `BusinessObjects.groovy` ，所以，类定义中没有包含的代码被编译到一个名为 `BusinessObjects` 的 .class 文件。

反编译

反编译这些类可能会非常有趣。由于 Groovy 处于代码顶层，所以生成的 .java 文件可能相当巨大；不过，您应当注意的是 Groovy 脚本中声明的类（如 `Person` ）与类之外的代码（比如 `BusinessObjects.class` 中找到的代码）之间的区别。在 Groovy 文件中定义的类完成了 `GroovyObject` 的实现，而在类之外定义的代码则被绑定到一个扩展自 `Script` 的类。

例如，如果研究由 BusinessObjects.class 生成的 .java 文件，可以发现：它定义了一个 `main()` 方法和一个 `run()` 方法。不用惊讶， `run()` 方法包含我编写的、用来创建这些对象的新实例的代码，而 `main()` 方法则调用 `run()` 方法。

这个细节的全部要点再一次回到了：对 Groovy 的理解越好，就越容易把它集成到 Java 程序中。有人也许会问："为什么我要这么做呢？"好了，我们想说您用 Groovy 开发了一些很酷的东西；那么如果能把这些东西集成到 Java 程序中，那不是很好吗？

只是为了讨论的原因，我首先试图 用 Groovy 创建一些有用的东西，然后我再介绍把它嵌入到普通 Java 程序中的各种方法。

再制作一个音乐 Groovy

我热爱音乐。实际上，我的 CD 收藏超过了我计算机图书的收藏。多年以来，我把我的音乐截取到不同的计算机上，在这个过程中，我的 MP3 收藏乱到了这样一种层度：只是表示品种丰富的音乐目录就有一大堆。

最近，为了让我的音乐收藏回归有序，我采取了第一步行动。我编写了一个快速的 Groovy 脚本，在某个目录的 MP3 收藏上进行迭代，然后把每个文件的详细信息（例如艺术家、专辑名称等）提供给我。脚本如清单 3 所示：

清单 **3.** 一个非常有用的 **Groovy** 脚本

```
package com.vanward.groovy
import org.farng.mp3.MP3File
import groovy.util.AntBuilder
class Song {

  mp3file
  Song(String mp3name){
    mp3file = new MP3File(mp3name)
  }
  getTitle(){
    mp3file.getID3v1Tag().getTitle()
  }
  getAlbum(){
    mp3file.getID3v1Tag().getAlbum()
  }
  getArtist(){
    mp3file.getID3v1Tag().getArtist()
  }
  String toString(){
    "Artist: " + getArtist() + " Album: " +
      getAlbum() + " Song: " + getTitle()
  }
  static getSongsForDirectory(sdir){
    println "sdir is: " + sdir
    ant = new AntBuilder()
    scanner = ant.fileScanner {
       fileset(dir:sdir) {
         include(name:"**/*.mp3")
       }
    }
    songs = []
    for(f in scanner){
      songs << new Song(f.getAbsolutePath())
    }
    return songs
  }
}
songs = Song.getSongsForDirectory(args[0])
songs.each{
 println it
}
```

正如您所看到的，脚本非常简单，对于像我这样的人来说特别有用。而我要做的全部工作只是把某个具体的目录名传递给它，然后我就会得到该目录中每个 MP3 文件的相关信息（艺术家名称、歌曲名称和专辑）。

现在让我们来看看，如果要把这个干净的脚本集成到一个能够通过数据库组织音乐甚至播放 MP3 的普通 Java 程序中，我需要做些什么。

Class 文件是类文件

正如前面讨论过的，我的第一个选项可能只是用 `groovyc` 编译脚本。在这个例子中，我期望 `groovyc` 创建 至少 两个 .class 文件 —— 一个用于 `Song` 类，另一个用于 `Song` 声明之后的脚本代码。

实际上， `groovyc` 可能创建 5 个 .class 文件。这是与 `Songs.groovy` 包含三个闭包有关，两个闭包在 `getSongsForDirectory()` 方法中，另一个在脚本体中，我在脚本体中对 `Song` 的集合进行迭代，并调用 `println` 。

因为 .class 文件中有三个实际上是 Song.class 和 Songs.class 的内部类，所以我只需要把注意力放在两个 .class 文件上。Song.class 直接映射到 Groovy 脚本中的 `Song` 声明，并实现了 `GroovyObject` ，而 Songs.class 则代表我在定义 `Song` 之后编写的代码，所以也扩展了 `Script` 。

此时此刻，关于如何把新编译的 Groovy 代码集成到 Java 代码，我有两个选择：可以通过 Songs.class 文件中的 `main()` 方法运行代码 （因为它扩展了 `Script` ），或者可以将 Song.class 包含到类路径中，就像在 Java 代码中使用其他对象一样使用它。

变得更容易些

通过 `java` 命令调用 Songs.class 文件非常简单，只要您记得把 Groovy 相关的依赖关系和 Groovy 脚本需要的依赖关系包含进来就可以。把 Groovy 需要的类全都包含进来的最简单方法就是把包含全部内容的 Groovy 可嵌入 jar 文件添加到类路径中。在我的例子中，这个文件是 groovy-all-1.0-beta-10.jar。要运行 Songs.class，需要记得包含将要用到的 MP3 库（jid3lib-0.5.jar>），而且因为我使用 `AntBuilder` ，所以我还需要在类路径中包含 `Ant` 。清单 4 把这些放在了一起：

清单 4. 通过 Java 命令行调用 Groovy

```
c:\dev\projects>java -cp  ./target/classes/;c:/dev/tools/groovy/
  groovy-all-1.0-beta-10.jar;C:/dev/tools/groovy/ant-1.6.2.jar;
  C:/dev/projects-2.0/jid3lib-0.5.jar
  com.vanward.groovy.Songs c:\dev09\music\mp3s
Artist: U2 Album: Zooropa Song: Babyface
Artist: James Taylor Album: Greatest Hits Song: Carolina in My Mind
Artist: James Taylor Album: Greatest Hits Song: Fire and Rain
Artist: U2 Album: Zooropa Song: Lemon
Artist: James Taylor Album: Greatest Hits Song: Country Road
Artist: James Taylor Album: Greatest Hits Song: Don't Let Me
  Be Lonely Tonight
Artist: U2 Album: Zooropa Song: Some Days Are Better Than Others
Artist: Paul Simon Album: Graceland Song: Under African Skies
Artist: Paul Simon Album: Graceland Song: Homeless
Artist: U2 Album: Zooropa Song: Dirty Day
Artist: Paul Simon Album: Graceland Song: That Was Your Mother
```

把 Groovy 嵌入 Java 代码

虽然命令行的解决方案简单有趣，但它并不是所有问题的最终解决方案。如果对更高层次的完善感兴趣，那么可能将 MP3 歌曲工具直接导入 Java 程序。在这个例子中，我想导入 Song.class ，并像在 Java 语言中使用其他类那样使用它。类路径的问题与上面相同 ：我需要确保包含了 *uber-Groovy* jar 文件、 `Ant` 和 jid3lib-0.5.jar 文件。在清单 5 中，可以看到如何将 Groovy MP3 工具导入简单的 Java 类中：

清单 5. 嵌入的 Groovy 代码

```
package com.vanward.gembed;
import com.vanward.groovy.Song;
import java.util.Collection;
import java.util.Iterator;
public class SongEmbedGroovy{
 public static void main(String args[]) {
  Collection coll = (Collection)Song.getSongsForDirectory
    ("C:\\music\\temp\\mp3s");
  for(Iterator it = coll.iterator(); it.hasNext();){
    System.out.println(it.next());
  }
 }
}
```

Groovy 类加载器

就在您以为自己已经掌握全部的时候，我要告诉您的是，还有更多在 Java 语言中使用 Groovy 的方法。除了通过直接编译把 Groovy 脚本集成到 Java 程序中的这个选择之外，当我想直接嵌入脚本时，还有其他一些选择。

例如，我可以用 Groovy 的 `GroovyClassLoader` ，动态地 加载一个脚本并执行它的行为，如清单 6 所示：

清单 **6. GroovyClassLoader** 动态地加载并执行 **Groovy** 脚本

```
package com.vanward.gembed;
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import groovy.lang.MetaMethod;
import java.io.File;
public class CLEmbedGroovy{
 public static void main(String args[]) throws Throwable{

  ClassLoader parent = CLEmbedGroovy.class.getClassLoader();
  GroovyClassLoader loader = new GroovyClassLoader(parent);

  Class groovyClass = loader.parseClass(
    new File("C:\\dev\\groovy-embed\\src\\groovy\\
      com\\vanward\\groovy\\Songs.groovy"));

  GroovyObject groovyObject = (GroovyObject)
    groovyClass.newInstance();

  Object[] path = {"C:\\music\\temp\\mp3s"};
  groovyObject.setProperty("args", path);
  Object[] argz = {};

  groovyObject.invokeMethod("run", argz);

 }
}
```

Meta，宝贝

如果您属于那群疯狂的人中的一员，热爱反射，喜欢利用它们能做的精彩事情，那么您将热衷于 Groovy 的 `Meta` 类。就像反射一样，使用这些类，您可以发现 `GroovyObject` 的各个方面（例如它的方法），这样就可以实际地 创建 新的行为并执行它。而且，这是 Groovy 的核心 —— 想想运行脚本时它将如何发威吧！

注意，默认情况下，类加载器将加载与脚本名称对应的类 —— 在这个例子中是 Songs.class，而不是 Song.class>。因为我（和您）知道 Songs.class 扩展了 Groovy 的 Script 类，所以不用想也知道接下来要做的就是执行 `run()` 方法。

您记起，我的 Groovy 脚本也依赖于运行时参数。所以，我需要恰当地配置 `args` 变量，在这个例子中，我把第一个元素设置为目录名。

更加动态的选择

对于使用编译好的类，而且，通过类加载器来动态加载 `GroovyObject` 的替代，是使用 Groovy 优美的 `GroovyScriptEngine` 和 `GroovyShell` 动态地执行 Groovy 脚本。

把 `GroovyShell` 对象嵌入普通 Java 类，可以像类加载器所做的那样动态执行 Groovy 脚本。除此之外，它还提供了大量关于控制脚本运行的选项。在清单 7 中，可以看到 `GroovyShell` 嵌入到普通 Java 类的方式：

清单 **7.** 嵌入 **GroovyShell**

```
package com.vanward.gembed;
import java.io.File;
import groovy.lang.GroovyShell;
public class ShellRunEmbedGroovy{
 public static void main(String args[]) throws Throwable{

  String[] path = {"C:\\music\\temp\\mp3s"};
  GroovyShell shell = new GroovyShell();
  shell.run(new File("C:\\dev\\groovy-embed\\src\\groovy\\
    com\\vanward\\groovy\\Songs.groovy"),
    path);
 }
}
```

可以看到，运行 Groovy 脚本非常容易。我只是创建了 `GroovyShell` 的实例，传递进脚本名称，然后调用 `run()` 方法。

还可以做其他事情。如果您喜欢，那么也可以得到自己脚本的 `Script` 类型的 `GroovyShell` 实例。使用 `Script` 类型，您就可以传递进一个 `Binding` 对象，其中包含任何运行时值，然后再继续调用 `run()` 方法，如清单 8 所示：

清单 **8.** 有趣的 **GroovyShell**

```
package com.vanward.gembed;
import java.io.File;
import groovy.lang.Binding;
import groovy.lang.GroovyShell;
import groovy.lang.Script;
public class ShellParseEmbedGroovy{
 public static void main(String args[]) throws Throwable{
  GroovyShell shell = new GroovyShell();
  Script scrpt = shell.parse(
    new File("C:\\dev\\groovy-embed\\src\\groovy\\
      com\\vanward\\groovy\\Songs.groovy"));

  Binding binding = new Binding();
  Object[] path = {"C:\\music\\temp\\mp3s"};
  binding.setVariable("args",path);
  scrpt.setBinding(binding);

  scrpt.run();
 }
}
```

Groovy 的脚本引擎

`GroovyScriptEngine` 对象动态运行脚本的时候，非常像 `GroovyShell`。区别在于：对于 `GroovyScriptEngine`，您可以在实例化的时候给它提供一系列目录，然后让它根据要求去执行多个脚本，如清单 9 所示：

清单 9. GroovyScriptEngine 的作用

```
package com.vanward.gembed;
import java.io.File;
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;
public class ScriptEngineEmbedGroovy{
 public static void main(String args[]) throws Throwable{

  String[] paths = {"C:\\dev\\groovy-embed\\src\\groovy\\
    com\\vanward\\groovy"};
  GroovyScriptEngine gse = new GroovyScriptEngine(paths);
  Binding binding = new Binding();
  Object[] path = {"C:\\music\\temp\\mp3s"};
  binding.setVariable("args",path);

  gse.run("Songs.groovy", binding);
  gse.run("BusinessObjects.groovy", binding);
 }
}
```

在清单 9 中，我向实例化的 `GroovyScriptEngine` 传入了一个数组，数据中包含我要处理的路径，然后创建大家熟悉的 `Binding` 对象，然后再执行仍然很熟悉的 `Songs.groovy` 脚本。只是为了好玩，我还执行了 `BusinessObjects.groovy` 脚本，您或许还能回忆起来，它在开始这次讨论的时候出现过。

Bean 脚本框架

最后，当然并不是最不重要的，是来自 Jakarta 的古老的 Bean 脚本框架（Bean Scripting Framework —— BSF）。BSF 试图提供一个公共的 API，用来在普通 Java 应用程序中嵌入各种脚本语言（包括 Groovy）。这个标准的、但是有争议的最小公因子方法，可以让您毫不费力地将 Java 应用程序嵌入 Groovy 脚本。

还记得前面的 `BusinessObjects` 脚本吗？在清单 10 中，可以看到 BSF 可以多么容易地让我把这个脚本插入普通 Java 程序中：

清单 **10. BSF** 开始工作了

```
package com.vanward.gembed;
import org.apache.bsf.BSFManager;
import org.codehaus.groovy.runtime.DefaultGroovyMethods;
import java.io.File;
import groovy.lang.Binding;
public class BSFEmbedGroovy{
 public static void main(String args[]) throws Exception {
  String fileName = "C:\\dev\\project\\src\\groovy\\
    com\\vanward\\groovy\\BusinessObjects.groovy";
  //this is required for bsf-2.3.0
  //the "groovy" and "gy" are extensions
  BSFManager.registerScriptingEngine("groovy",
    "org.codehaus.groovy.bsf.GroovyEngine", new
      String[] { "groovy" });
  BSFManager manager = new BSFManager();
  //DefaultGroovyMethods.getText just returns a
  //string representation of the contents of the file
  manager.exec("groovy", fileName, 0, 0,
    DefaultGroovyMethods.getText(new File(fileName)));
  }
}
```

结束语

如果在本文中有一件事是清楚的话，那么只件事就是 Groovy 为了 Java 代码内部的重用提供了一堆选择。从把 Groovy 脚本编译成普通 Java .class 文件，到动态地加载和运行脚本，这些问题需要考虑的一些关键方面是灵活性和耦合。把 Groovy 脚本编译成普通 .class 文件是 使用 您打算嵌入的功能的最简单选择，但是动态加载脚本可以使添加或修改脚本的行为变得更容易，同时还不必在编译上牺牲时间。（当然，这个选项只是在接口不变的情况下才有用。）

把脚本语言嵌入普通 Java 不是每天都发生，但是机会确实不断出现。这里提供的示例把一个简单的目录搜索工具嵌入到基于 Java 的应用程序中，这样 Java 应用程序就可以很容易地变成 MP3 播放程序或者其他 MP3 播放工具。虽然我 可以 用 Java 代码重新编写 MP3 文件搜索器，但是我不需要这么做：Groovy 极好地兼容 Java 语言，而且，我很有兴趣去摆弄所有选项！

实战 Groovy: 用 Groovy 生成器作标记

抛开标记语言的细节，聚焦应用程序的内容

Groovy 生成器让您能够利用诸如 Swing 这样的框架来模拟标记语言（如 XML、HTML、Ant） 任务以及 GUI。它们对于快速原型化非常有 用，并且正像 Andrew Glover 这个月在“实战 *Groovy*”专栏中向您展示的那样，当您马上需要可消费的标记时，它们是数据绑定框架的一种便利的替代方案。

几个月前，当我最初撰写有关 实战 Groovy: 用 Groovy 进行 Ant 脚本编程的文章时，我提及了 Groovy 中的 生成器概念。在那篇文章里，我向您展示了，使用一个叫做 `AntBuilder` 的 Groovy 类，构建富有表现力的 Ant 构建文件是多么容易。本文中，我将深入 Groovy 生成器的世界，向您展示您还能用这些强大的类做些什么。

用生成器进行构建

Groovy 生成器让您能够利用诸如 Swing 这样的框架来模拟标记语言（如 XML、HTML、Ant） 任务以及 GUI。使用生成器，您可以迅速地创 建复杂的标记（如 XML），而无须理会 XML 本身。

生成器的范例非常简单。生成器的实例的方法表示该标记（如 HTML 中的 `<body>` 标 签）的元素。方法的创建于闭包中的对象表示子节点（例如， `<body>` 标签中所包含的 `<p>` 标签）。

为了便于您查看这一过程，我将创建一个简单的生成器，以程序方式来表示一个具有清单 1 所示结构的 XML 文档。

关于本系列文章

在开发实践中采用任何工具的关键是，了解何时使用这些工具，何时将其弃而不用。脚本语言可能是对工具包的一个功能强大的扩充，但是只有 在正确应用于适当的环境时才是如此。总之，实战 *Groovy*系列文章旨在展示 Groovy 的实际使用，以及何时和如何成功应用它。

清单 **1.** 简单 **XML** 结构

```
<person>
  <name first="Megan" last="Smith">
    <age>32</age>
    <gender>female</gender>
  </name>
  <friends>
    <friend>Julie</friend>
    <friend>Joe</friend>
    <friend>Hannah</friend>
  </friends>
</person>
```

要表示这个结构非常简单。首先将 person 方法连接到生成器实例，现在它表示 XML 的根节点，即 <person> 。要创建子节点，我创建一个闭包并声明一个名叫 name 的新对象（它接收 map 形式的参数。顺便说一下，这些参数是元素的属性的基础。

接下来，在 name 对象中，将两个附加对象连接到闭包，一个对象是 age ，另一个是 gender ，它们对应于 <name> 的类似子元素。您明白其中的诀窍了么？确实很简单。

<friends> 元素是 <person> 的兄弟元素，于是我跳出这个闭包，声明了一个 friends 对象，当然，还附加了一个集合了多个 friend 元素的闭包，如清单 2 所示。

清单 **2.** 生成器是如此的简单

```
import groovy.xml.*
import java.io.*
class XMLBuilder{

 static void main(args) {

   writer = new StringWriter()
   builder = new MarkupBuilder(writer)
   friendnames = [ "Julie", "Joey", "Hannah"]

    builder.person() {
      name(first:"Megan", last:"Smith") {
        age("33")
        gender("female")
      }
      friends() {
        for (e in friendnames) { friend(e) }
      }
   }
   println writer.toString()
 }
}
```

如您所见，这里的 Groovy 表示非常优雅，且易于映射到相应的标记表示。在底层，Groovy 显然在处理烦人的标记元素（如 < and >），使我们可以将更多精力放在内容上，而不必过分在意结构的细节。

显示 HTML

生成器也可以有助于构建 HTML，这在开发 Groovlet 时可以派上用场。如同小菜一碟，假设我要创建一个如清单 3 所示的 HTML 页面。

清单 3. HTML 101

```
<html>
 <head>
  <title>Groov'n with Builders</title>
 </head>
 <body>
  <p>Welcome to Builders 101\. As you can see this Groovlet is fai
 </body>
</html>
```

我可以轻易地将它编码在 Groovy 中，清单 4 所示。

清单 4. HTML in Groovy 101

```
import groovy.xml.*
import java.io.*
class HTMLBuilderExample{

 static void main(args) {
  writer = new StringWriter()
  builder = new MarkupBuilder(writer)

  builder.html(){
    head(){
      title("Groov'n with Builders"){}
    }
    body(){
      p("Welcome to Builders 101\. As you can see " +
        "this Groovlet is fairly simple.")
    }
  }
  println writer.toString()
}
```

来点有意思的，让我们再看看用生成器建立一个成熟的 GUI 有多么容易。前面我曾提到过，Groovy 的 `SwingBuilder` 使它能够以一种极为简单的方式构造 GUI。您可以查阅清单 5 中 `SwingBuilder` 是如何工作的。

清单 5. Groovy 中的 GUI 生成器真的很“GROOVY”（很“棒”）

```
import java.awt.FlowLayout
import javax.swing.*
import groovy.swing.SwingBuilder
class SwingExample{

 static void main(args) {
   swinger = new SwingBuilder()
   langs = ["Groovy", "Ruby", "Python", "Pnuts"]

    gui = swinger.frame(title:'Swinging with Groovy!', size:[290,1
     panel(layout:new FlowLayout()) {
       panel(layout:new FlowLayout()) {
         for (lang in langs) {
           checkBox(text:lang)
         }
       }
       button(text:'Groovy Button', actionPerformed:{
         swinger.optionPane(message:'Indubitably Groovy!')
             .createDialog(null, 'Zen Message').show()
       })
       button(text:'Groovy Quit', actionPerformed:{ System.exit(0)
    }
  }
  gui.show()
  }
}
```

图 1 显示了上面的结果，还不错吧？

图 1. Groovy 中神奇的 GUI 编程



可以想像，对于原型化，`SwingBuilder` 是一个多么强大的工具，不是么？

一些事实

这些例子虽然琐碎，却也有趣。我希望我能让您明白，Groovy 的生成器可以让您避免特定语言（如 XML）中的底层标记。显然，有时避免 XML 或 HTML 会更好，并且，那些标记协助器（facilitator）对 Java 平台来说并不陌生。例如，我最喜欢的 XML 协助框架是 JiBX。

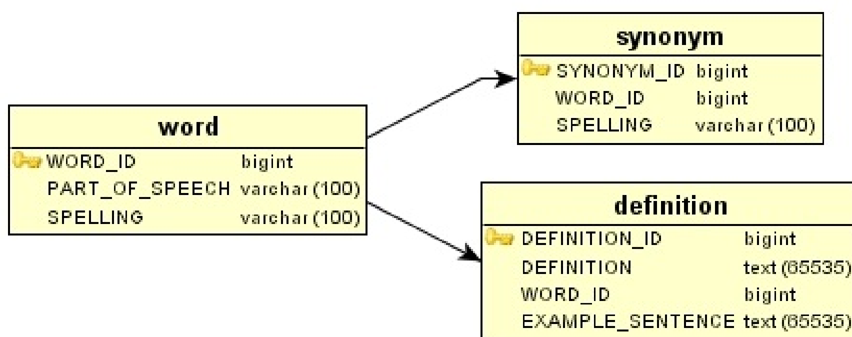
使用 JiBX，您可以轻易地将 XML 结构映射到对象模型，反之亦然。绑定是个强大的范例，有不计其数的类似工具拥有此功能，如 JAXB、 Castor 和 Zeus 等。

绑定框架的惟一缺点是，它们 恐怕要耗费不少时间。幸运的是，您可以使用 Groovy 的生成器作为一个 较简单的解决方案，这在某些情况下是有效的。

用生成器进行伪绑定

假设有一个英文词典的简单数据库。有一个表用于 `word` ，另一个表用于 `definition` ，最后还有一个表用于 `synonym` 。图 2 是这个数据库的简单表示。

图 **2.** 词典数据库



如您所见，这个数据库非常直观： `word` 与 `definition` 和 `synonym` 具有一对多的关系。

词典数据库拥有一个消费者，他在寻求一种表示数据库内容的关键方面的 XML 结构。所寻求的 XML 结构如清单 6 所示。

清单 **6.** 可采用的词典 **XML**

```
<words>
 <word spelling="glib" partofspeech="adjective">
   <defintions>
     <defintion>Performed with a natural, offhand ease.</defintion
     <defintion>Marked by ease and fluency of speech or writing th
     or stems from insincerity, superficiality, or deceitfulness</
   </defintions>
   <synonyms>
     <synonym spelling="artful"/>
     <synonym spelling="urbane"/>
   </synonyms>
 </word>
</words>
```

如果选择使用 JiBX 这样的绑定框架来解决这个问题，则很可能需要创建一些中间对象模型，以从关系模型到达最终的 XML 模型。然后必须将数据库内容读取到对象模型中，并请求底层框架将其内部的结构编组为 XML 格式。

这一过程内含了将对象结构映射到 XML 格式的步骤（使用所需的框架过程）。某些框架，如 JAXB，实际上是从 XML 和其他框架（如 JiBX ）生成 Java 对象，允许您自定义自己的 Java 对象到 XML 格式的映射。总之，这都需要大量的工作。

并且，这是一项宏伟的计划。我并不提倡避免使用绑定框架。这里，我要声明：我已经预先警告过您。我计划向您展示的是一个生成 XML 的便捷方式。

可消费的 XML 很简单

使用 Groovy 的 `MarkupBuilder` ，结合新的数据库访问框架 GroovySql，您可以轻易地生成可消费的 XML。您 所要做的只是计算出所需的查询，并将结果映射到生成器实例 —— 然后，您马上就可以得到表示词典数据库内容的 XML 文档。

让我们逐步来了解这一过程。首先，创建一个生成器实例，在本例中是 `MarkupBuilder` ，因为您想要生成 XML。最外面的 XML 元 素（也就是“根”）是 `words` ，这样就创建了一个 `words` 方法。在闭包里， 调用第一个查询，并在迭代中将查询结果映射到 `word` 子节点。

接着，通过两个新的查询，创建 `word` 的两个子节点。创建一个 `definitions` 对象，并在迭代中映射它，接着用同样的方法处理 `synonyms` 。

清单 **7.** 用生成器集合所有元素

```
import groovy.sql.Sql
import groovy.xml.MarkupBuilder
import java.io.File
import java.io.StringWriter
class WordsDbReader{
 static void main(args) {
   sql = Sql.newInstance("jdbc:mysql://localhost/words",
     "words", "words", "org.gjt.mm.mysql.Driver")
   writer = new StringWriter()
   builder = new MarkupBuilder(writer)
   builder.words() {
     sql.eachRow("select word_id, spelling, part_of_speech from wo
       builder.word(spelling:row.spelling, partofspeech:row.part_o

        builder.definitions(){
           sql.eachRow("select definition from definition where wo
             ${row.word_id}"){ defrow |
             builder.definition(defrow.definition)
           }
        }

        builder.synonyms(){
           sql.eachRow("select spelling from synonym where word_id
             synrow |
             builder.synonym(synrow.spelling)
           }
        }
      }
     }
   }
  new File("dbouuput.xml").withPrintWriter{ pwriter |
     pwriter.println writer.toString()
  }
}
}
```

结束语

这里，我向您展示的绑定解决方案似乎简单得让人难以置信，特别是以 Java 纯化论者的观点看来更是如此。尽管该解决方案不比使用绑定框架（如 JABX 和 JiBX）更好，但它确实更快一些 —— 而且，我主张使用这样较简单的方法。是不是我在_简单的_Java 代码中做一些类似的事情？是的，但我敢肯定，某些时候我也不得不处理 XML。

用 Groovy 生成器进行开发的速度和简易性，在调用标记的时候可大显神威。例如，就像在第二个例子里展示的那样，我可以马上加快数据库的 XML 表示。对于原型化，或者当需要以最少的开发时间和精力来产生可工作的解决方案时，生成器也是一个不错的选择。

在下个月的 实战 *Groovy*中我会讲些什么呢？哦，当然是在 Java 语言中使用 Groovy ！

实战 Groovy: 用 Groovy 打造服务器端

用 Groovlet 和 GSP 进行动态服务器端编程

Groovlet 和 GroovyServer Pages（GSP）框架都是建立在 Java Servlet API 基础之上。不过，与 Strut 和 JSF 不同，Groovy 的服务器端实现不意味着适用于所有情况。相反，它提供了一种快速而又方便地开发服务器端应用程序的简化方法。下面请跟随 Groovy 的鼓吹者 Andrew Glover，听听他如何介绍这些框架，并展示它们的应用。

Java 平台为自己赢得了服务器端应用程序开发的首选平台的名声。Servlet 是服务器端 Java 技术的强大支柱，因此有无数的框架是围绕着 Servlet API 建立 起来的，其中包括 Strut、JavaServer Faces (JSF) 和 Tapestry。您可能已经猜到，Groovy 也是以 Servlet API 为基础建立起来的框架，不过，这个框架的目的是简化开发。

Groovlet 和 GroovyServer Pages（GSP）框架的目的是提供一种优雅而又简单的平台，将它用于构建复杂程度不高的 Web 应用程序。就像 GroovySql 不是数据库开发的惟一选择一样，Groovlet 框架也不是 像 Strut 那样具有更丰富功能的框架的替代品。Groovlet 只是 开发人员寻求容易配置和产生工作代码的快速方法时的一种选择。

例如，不久前，我需要 —— 快速地 —— 提供一个 stub 应用程序，以测试像 `xml-rpc` API 这样的客户端。显然可以用一个 servlet 快速编写出所需要的功能，但是我从没想过钻研 Strut，一秒钟也没有。我考虑过使用基本的普通 Java Servlet API 编写 servlet 及其相关的逻辑，但是由于需要尽快地使用这项功能，所以我选择了使用 Groovlet 快速完成它。

很快您就可看到，这种选择是显而易见的。

在深入研究使用 Groovlet 进行编程之前，我想简单介绍一个在示例代码中会用到的 Groovy 特性。 在几个月前的 alt.lang.jre: 感受 Groovy 一文中，我第一次介绍了 `def` 关键字。

关于本系列

将任何工具添加到开发实践中的关键是了解什么时候使用它和什么时候 不使用它。脚本语言可以为您增加特别强大的工具，但前提是在相关的场景 中正确地使用它。因此，实战 *Groovy* 的系列文章专门探讨 Groovy 的实际使用，并指导读者什么时候 以及如何成功地使用这些工具。

在脚本中定义函数

在普通 Java 编程中，方法必须存在于类对象中。事实上，所有行为都必须在类的上 下文中定义。不过在 Groovy 中，行为可以在函数 中定义，而函数可以在类定义之外定义 。

这些函数可以直接用名称引用，并且可以在 Groovy 脚本中定义，这样非常 有助于它们的重复使用。Groovy 函数需要 `def` 关键字， 可以将关键字想像为在脚本范围内可用的全局静态方法。因为 Groovy 是动态类型的语言，所以 `def` 不需要对参数作任何类型声明， `def` 也不需要 `return` 语句。

例如，在清单 1 中，我定义了一个简单的函数，它将输出一个集合的内容，而不管这个集合是 `list` 还是 `map` 。然后我定义 一个 `list` ，填充它，并调用我新定义的 `def` 。之后，我创建一个 `map` ，并对这个集合做了同样的操作。

清单 **1.** 这就是 **def!**

```
def logCollection(coll){
  counter = 0;
  coll.each{ x |
    println "${++counter} item: ${x}"
  }
}
lst = [12, 3, "Andy", 'c']
logCollection(lst)
mp = ["name" : "Groovy", "date" : new Date()]
logCollection(mp)
```

`def` 不需要 `return` 语句，因此如果最后一行产生某个 值，那么这个值由 `def` 返回。例如，在清单 2 中，代码定义了一个 `def` ，它返回传递进来的变量的名称。 我可以编写它，让它带有或者不带 `return` 语句，得到的结果是相同的。

清单 **2.** 在 **def** 中 **return** 语句是可选的

```
def getJavaType(val){
  val.class.getName()
}
tst = "Test"
println getJavaType(tst)
```

在编写简单的脚本时， `def` 关键字会非常好用。您很快就会看到，在开发 Groovlet 时，这个关键字也会派上用场。

Groovlet 和 GSP

使用 Groovlet 和 GSP 的前提条件相当简单：需要一个 servlet 容器，以及最新、最伟大 版本的 Groovy。这些框架的好处是它们通过一个 web.xml 文件将所选模式的所有 URL 映射到特定的 servlet。因此， 建立 Groovlet 和 GSP 的实现的第一步是定义一个 Web 应用程序上下文，并更新它的相关 web.xml 文件。 这个文件将包括特定的 servlet 类定义以及它们对应的 URL 模式。

我将使用 Apache Jakarta Tomcat，并且已创建了一个名为 *groove* 的上下文。目录结构如清单 3 所示：

清单 **3. groove** 上下文的目录列表

```
./groove:
drwxrwxrwx+   3 aglover  users        0 Jan 19 12:14 WEB-INF
./WEB-INF:
-rwxrwxrwx+   1 aglover  users      906 Jan 16 14:37 web.xml
drwxrwxrwx+   2 aglover  users        0 Jan 19 17:12 lib
./WEB-INF/lib:
-rwxrwxrwx+   1 aglover  users   832173 Jan 16 14:28 groovy-1.0-bet
-rwxrwxrwx+   1 aglover  users    26337 Jan 16 14:29 asm-1.5.2.jar
```

在 WEB-INF 目录中要有一个 web.xml 文件，它至少有一个清单 4 中的元素：

清单 **4.** 一个完全配置的 **web.xml** 文件

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
    "http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <servlet>
      <servlet-name>GroovyServlet</servlet-name>
      <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>GroovyTemplate</servlet-name>
        <servlet-class>groovy.servlet.TemplateServlet</servlet-clas
    </servlet>
    <servlet-mapping>
        <servlet-name>GroovyServlet</servlet-name>
        <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>GroovyTemplate</servlet-name>
        <url-pattern>*.gsp</url-pattern>
    </servlet-mapping>
</web-app>
```

上述 web.xml 文件中的定义声明了以下内容：所有以 `.groovy` 结尾的请求（如 `http://localhost:8080/groove/hello.groovy` ）都将发送给 类 `groovy.servlet.GroovyServlet` ， 而所有以 `.gsp` 结尾的请求 都将送给类 `groovy.servlet.TemplateServlet` 。

下一步是将两个 jar 放到 lib 目录中：groovy 发行版本的 jar（在这里是 groovy-1.0-beta-9.jar）和 对应的 `asm` jar （对于 groovy beta-9 来说是 asm-1.5.2.jar）。

瞧，就是这么简单 —— 我已经准备好了。

Groovlet，请出场

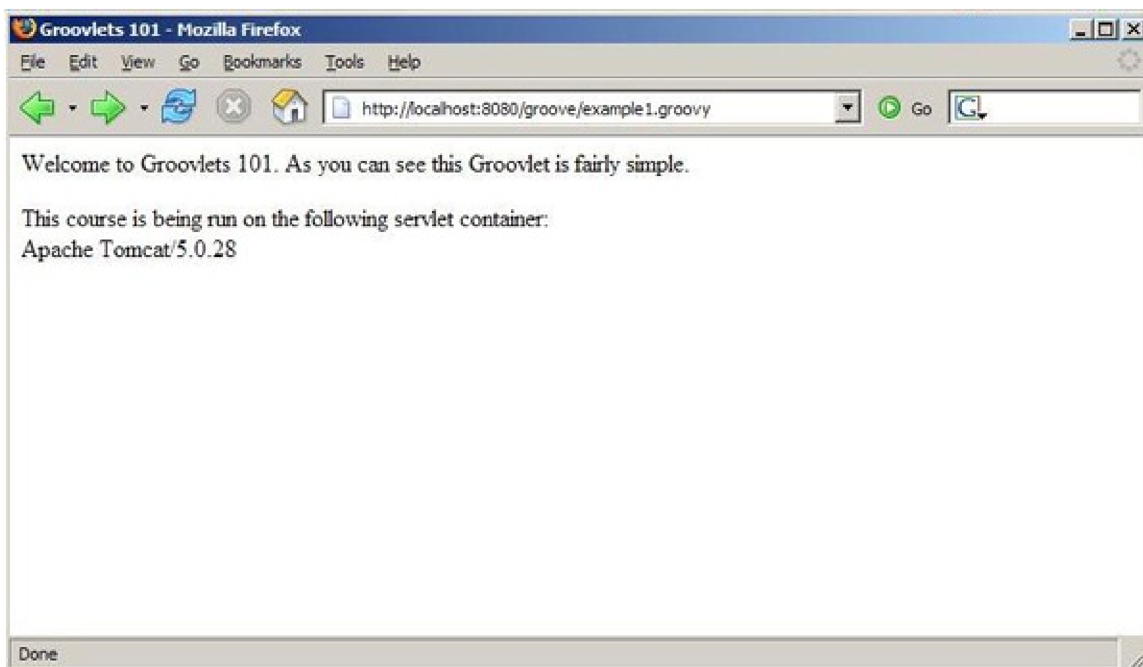
编写 Groovlet 无疑很简单，因为 Groovy 只有很少的几个 对类继承扩展的要求。使用 Groovlet 不需要扩展 `javax.servlet.http.HttpServlet` 、 `javax.servlet.GenericServlet` 或者一些华而不实的 `GroovyServlet` 类。事实上，创建 Groovlet 就像创建一个 Groovy 脚本一样简单。甚至不必 创建一个类。在清单 5 中，我编写了一个简单的 Groovlet，它 做两件事：打印一些 HTML，然后提供一些关于它所在的容器的信息。

清单 5. 开始使用 Groovlet

```
println """
<html><head>
<title>Groovlets 101</title>
</head>
<body>
<p>
Welcome to Groovlets 101\. As you can see
this Groovlet is fairly simple.
</p>
<p>
This course is being run on the following servlet container: </br>
${application.getServerInfo()}
</p>
</body>
</html>
"""
```

如果在浏览器中观看这个 Groovy，它看起来与图 1 所示类似。

图 **1.** 简单 **Groovlet** 的输出



仔细观察清单 5 中的 Groovlet，会让您回想起第一次编写 Groovy 脚本的时候。 首先，没有 `main` 方法或者类定义，只有一些简单的代码。而且，Groovlet 框架隐式地提供实例变量，比如 `ServletRequest`、`ServletResponse`、`ServletContext` 和 `HttpSession`。注意我是如何通过 `application` 变量引用 `ServletContext` 的实例的。如果想获得 `HttpSession` 的实例，那么就要使用 `session` 变量名。与此类似，可以对 `ServletRequest` 和 `ServletResponse` 分别使用 `request` 和 `response`。

一个诊断 Groovlet

编写 Groovlet 不仅像创建一个 Groovy 脚本那样简单，而且还可以用 `def` 关键字定义函数，并在 Groovlet 中直接调用它们。为了展示这一点，我将 创建一个非凡的 Groovlet，它将对 Web 应用程序进行一些诊断。

假设您编写了一个 Web 应用程序，它被世界上不同的客户所购买。您有一个 大客户群，并且不断发布这个应用程序有一段时间了。从过去的支持问题中，您 注意到许多急切的客户电话都与错误的 JVM 版本和错误的对象关系映射（ORM）所导致的问题有关。

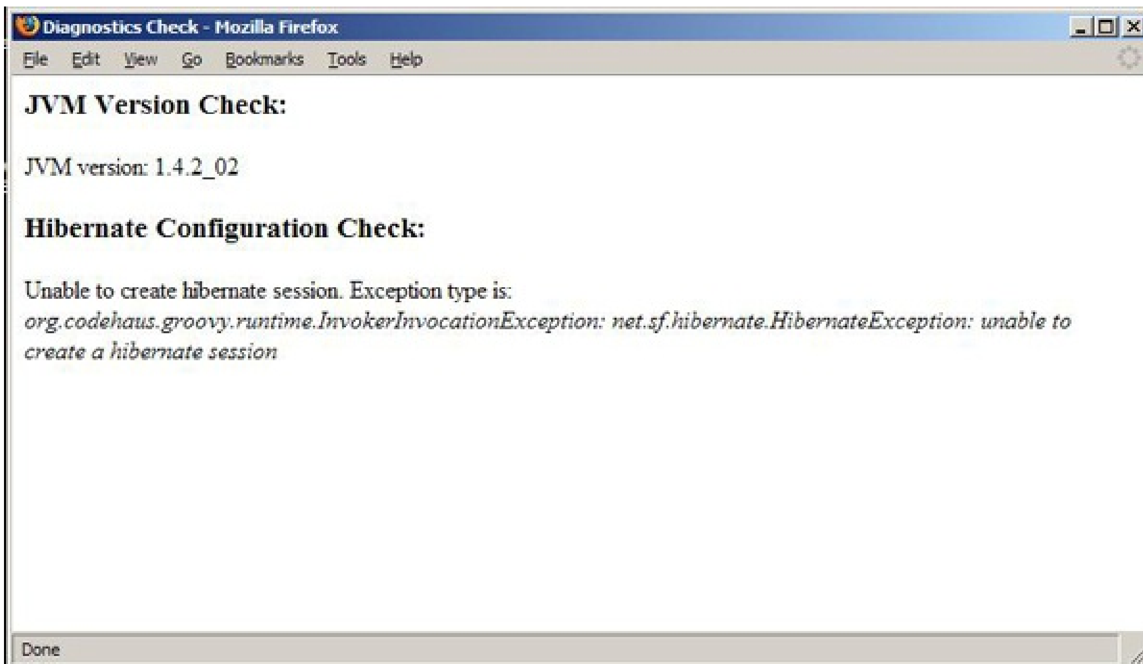
您很忙，所以让我拿出一个解决方案。我用 Groovlet 迅速地 创建了一个简单的 诊断脚本，它将验证 VM 版本，并试图创建一个 Hibernate 会话（请参阅参考资料）。我首先创建 两个函数，并在浏览器连接脚本时调用它们。 清单 6 定义了这个诊断 Groovlet：

清单 6. 一个诊断 Groovlet

```
import com.vanward.resource.hibernate.factory.DefaultHibernateSessi
/**
 * Tests VM version from environment- note, even 1.5 will
 * cause an assertion error.
 */
def testVMVersion(){
  println "<h3>JVM Version Check: </h3>"
  vers = System.getProperty("java.version")
  assert vers.startsWith("1.4"): "JVM must be at least 1.4"
  println "<p>JVM version: ${vers} </p>"
}
/**
 * Attempts to create an instance of a hibernate session. If this
 * works we have a connection to a database; additionally, we
 * have a properly configured hibernate instance.
 */
def testHibernate(){
  println "<h3>Hibernate Configuration Check: </h3>"
  try{
    sessFactory = DefaultHibernateSessionFactory.getInstance()
    session = sessFactory.getHibernateSession()
    assert session != null: "Unable to create hibernate session.
    Session was null"
    println "<p>Hibernate configuration check was successful</p>"
  }catch(Throwable tr){
    println """
    <p>Unable to create hibernate session. Exception type is: <br/>
    <i>${tr.toString()} </i><br/>
    </p>
    """
  }
}
println """
<html><head>
<title>Diagnostics Check</title></head>
<body>
"""
testVMVersion()
testHibernate()
println """
</body></html>
"""
```

这个 Groovlet 的验证逻辑非常简单，但是它可以完成这项工作。只要将 诊断脚本绑定到 web 应用程序即可，当客户服务台收到电话时，它们将指点客户用浏览器访问 `Diagnostics.groovy` 脚本，并让这些脚本报告它们的发现。 结果可能看起来像图 2 这样。

图 **2.** 诊断 **Groovlet** 的输出



那些 GSP 呢？

到目前为止，我主要关注于编写 Groovlet。不过，正如您将会看到的那样， 很容易用 Groovy 的 GSP 页对 Groovlets 框架进行补充，就像 JSP 补充 Servlet API 一样。

表面上，GSP 看起来很像 JSP，实际上它们不可能有太大的差别，因为 GSP 框架其实就是一个模板引擎。如果不熟悉模板引擎，那么可能需要快速地 回顾一下上月的文章。

虽然 GSP 和 JSP 是根本不同的技术，但是 GSP 是加入表达 Web 应用程序的视图的很 好候选人，这一点它们是类似的。您可能会想起来，在上月的文章中，有一项促进视图的技术可以将应用程序的业务逻辑问题与其相应的视图分离。 如果快速查看一下清单 6 中的诊断 Groovlet，就可以看出 GSP 代码改进了哪些地方。

是的，Groovlet 有些简陋，不是吗？问题在于它混合了应用程序逻辑和大量输出 HTML 的 `println` 。幸运的是，可以通过创建一个简单的 GSP 来补充这个 Groovlet，从而解决这个问题。

示例 GSP

创建 GSP 与创建 Groovlet 一样容易。GSP 开发的关键是认识到 GSP 实质上 是一个模板，因此，它最适合有限的逻辑。我将在清单 7 中创建一个 简单的 GSP 作为开始：

清单 7. 一个简单的 GSP

```
<html>
<head><title>index.gsp</title></head>
<body>
<b><% println "hello gsp" %></b>
<p>
<% wrd = "Groovy"
   for (i in wrd){
 %>
 <h1> <%=i%> <br/>

 <%} %>
</p>
</body>
</html>
```

观察上面的 GSP 可能很容易让您回想起标准 Groovy 模板开发。 像 JSP 一样，它使用 `<%` ，但是， 与 Groovlet 框架类似，它允许您访问常用 servlet 对象，比如 `ServletRequest` 、 `ServletResponse` 、 `ServletContext` 和 `HttpSession` 对象。

重构应用程序 ...

在练习编程语言或者平台发展的时候，重构老的代码可以学到很多东西。我将重构一月份专栏中的简单报告应用程序，那时候您才刚开始学习 GroovySql。

您还记得吗，我构建了一个快速但不完善的报告应用程序，它可以在组织中有 多次使用。但结果是，它变成了研究公司数据库活动的相当流行的应用程序。 现在，非技术人员希望可以访问这个巨大的报告，但是他们不想很费事地在自已的 计算机上安装 Groovy 来运行它。

我多少预计到了这种情况的发生，解决方案实际上 是显而易见的：让报告应用程序支持 Web。很幸运，Groovlet 和 GSP 使重构变成小事一桩。

重构报告应用程序

首先，我将处理 > GroovySql 一文的清单 12 中的简单应用程序。重构这个应用程序很容易：只要将所有 `println` 替换成用 `setAttribute()` 方法，然后将实例变量放入 `HttpRequest` 对象的逻辑中即可。

下一步是用 `RequestDispatcher` 将 `request` 转发给 GSP，它会处理报告应用程序的视图部分。清单 8 定义了新的报告 Groovlet：

清单 **8.** 重构后的数据库报告应用程序

```
import groovy.sql.Sql
/**
 * forwards to passed in page
 */
def forward(page, req, res){
  dis = req.getRequestDispatcher(page);
  dis.forward(req, res);
}
sql = Sql.newInstance("jdbc:mysql://yourserver.anywhere/tiger", "sc
        "tiger", "org.gjt.mm.mysql.Driver")

uptime = null
questions = null
insertnum = null
selectnum = null
updatenum = null
sql.eachRow("show status"){ status |
  if(status.variable_name == "Uptime"){
         uptime =  status[1]
     request.setAttribute("uptime", uptime)
  }else if (status.variable_name == "Questions"){
         questions =  status[1]
     request.setAttribute("questions", questions)
  }
}
request.setAttribute("qpm", Integer.valueOf(questions) /
Integer.valueOf(uptime) )
sql.eachRow("show status like 'Com_%'"){ status |
    if(status.variable_name == "Com_insert"){
         insertnum =  Integer.valueOf(status[1])
    }else if (status.variable_name == "Com_select"){
         selectnum =  Integer.valueOf(status[1])
    }else if (status.variable_name == "Com_update"){
          updatenum =  Integer.valueOf(status[1])
    }
}
request.setAttribute("qinsert", 100 * (insertnum / Integer.valueOf(
request.setAttribute("qselect", 100 * (selectnum / Integer.valueOf(
request.setAttribute("qupdate", 100 * (updatenum / Integer.valueOf(
forward("mysqlreport.gsp", request, response)
```

清单 8 中的代码应当是您相当熟悉的。我只是将以前应用程序中的所有 `println` 替换掉，并添加了 `forward` 函数来处理报告的视图部分。

添加视图部分

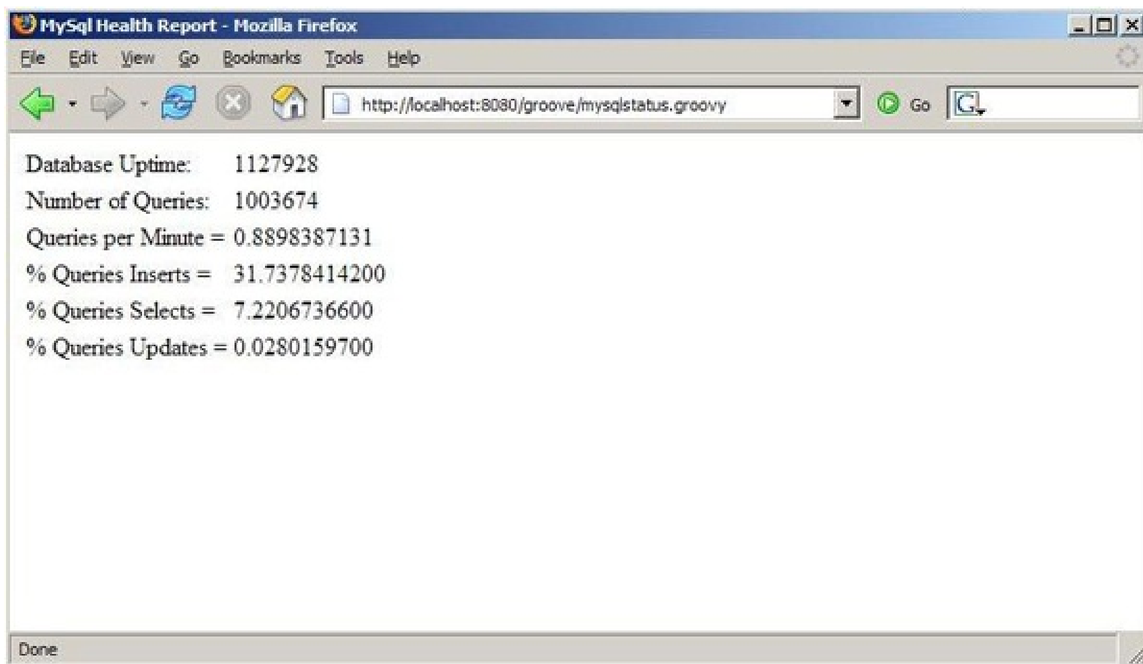
下一步是创建 GSP 来处理报告应用程序的视图。 因为我是工程师而不是一个艺术家，所以我的视图是相当 简单的 —— 一些 HTML 加上一个表，如清单 9 所示：

清单 **9.** 报告的视图部分

```
<html><head>
<title>MySql Health Report</title>
</head>
<body>
<table>
<tr>
  <td>Database Uptime:</td><td><% println
  "${request.getAttribute("uptime")}" %></td>
</tr>
<tr>
  <td>Number of Queries:</td><td><% println
  "${request.getAttribute("questions")}" %></td>
</tr>
<tr>
  <td>Queries per Minute =</td><td><% println
  "${request.getAttribute("qpm")}" %></td>
</tr>
<tr>
  <td>% Queries Inserts =</td><td><% println
  "${request.getAttribute("qinsert")}" %></td>
</tr>
<tr>
  <td>% Queries Selects =</td><td><% println
  "${request.getAttribute("qselect")}" %></td>
</tr>
<tr>
  <td>% Queries Updates =</td><td><% println
  "${request.getAttribute("qupdate")}" %></td>
</tr>
</table>
</body>
</html>
```

运行新的报告应当生成如图 3 所示的输出，数字会有变化。

图 **3.** 重构后的报告应用程序的输出



结束语

如您所见，当所需要的功能相当简单并且需要尽快完成时，Groovlet 和 GSP 是进行服务器端开发的当然之选。 这两个框架都特别灵活，并且其代码到视图的转化时间事实上是无可匹敌的。

不过，需要强调的是，Groovlet 不是 Strut 的替代品。GSP 框架不是直接 在速度上与其他产品竞争。GroovySql 不是 Hibernate 的替代品。而 Groovy 也不是 Java 语言的替代品。

无论如何，这些技术是补充，在大多数情况下，Groovy 是快速开发的更简单的一种选择。 就像 GroovySql 是直接使用 JDBC 的替代方法一样，Groovlet 和 GSP 实际上是直接使用 Servlet API 的替代品。

下个月，我将探讨 GroovyMarkup 的奇妙世界。

实战 Groovy: 使用 Groovy 模板进行 MVC 编程

使用 *Groovy* 模板引擎框架简化报表视图

视图是 MVC 编程的一个重要部分，而 MVC 编程本身又是企业应用程序开发的一个重要组件。在这篇实战 Groovy 的文章中，Andrew Glover 向您介绍了 Groovy 的模板引擎框架是如何用来简化视图编程的，并如何使您的代码更加经久容易维护。

在最近的 实战 *Groovy*系列中，我们已经介绍过 Groovy 是构建报表统计程序的一个非常好的工具。我们使用了一个校验和报表统计应用程序的例子向您介绍了 使用 Groovy 编写 Ant 脚本和一个数据库报表统计来展示 使用 GroovySql 进行 JDBC 编程。这两个示例统计报表都是在 Groovy 脚本中生成的，而且都具有一个相关的“视图”。

在校验和统计报表的例子中，视图代码有些混乱。更糟糕的是，它可能会对维护造成一些困难：如果我曾经希望修改视图的某个特定方面，就必须修改脚本中的代码。因此在本月的文章中，我将向您展示如何使用 Groovy 模板引擎和上一篇文章中的统计报表应用程序来对统计报表进行简化。

模板引擎与 XSLT 很类似，可以产生模板定义的任何格式，包括 XML、HTML、SQL 和 Groovy 代码。与 JSP 类似，模板引擎会简化将视图关注的内容分隔成单独的实体，例如 JSP 或模板文件。例如，如果一个报表希望的输出是 XML，那么您就可以创建一个 XML 模板，它可以包含一些占位符，在运行时替换为实际的值。模板引擎然后可以通过读取该模板并在这些占位符和运行时的值之间建立映射来实现对模板的转换。这个过程的输出结果是一个 XML 格式的文档。

关于本系列文章

在开发实践中采用任何工具的关键是了解何时使用这些工具，何时将其弃而不用。脚本语言可能是对工具包的一个功能强大的扩充，但是只有在正确应用于适当的环境时才是如此。总之，实战 *Groovy*系列文章是专门用来展示对 Groovy 的实际使用，以及何时和如何成功应用它。

在开始介绍 Groovy 模板之前，我想首先来回顾一下 Groovy 中 `String` 类型的概念。所有的脚本语言都试图使字符串的使用变得异常简单。再次声明，Groovy 并不是让我们的技能逐步退化。在开始之前，请点击本页面顶部或底部的 **Code**图标（或者参阅 下载一节的内容），下载本文的源代码。

不能获得充足的 Strings

不幸的是，Java™ 代码中的 Strings 非常有限，不过 Java 5.0 应用程序承诺将有一些引人注目的新特性。现在，Groovy 解决了两个最差的 Java `String` 限制，简化了编写多行字符串和进行运行时替换的功能。有一些简单的例子可以对 Groovy `String` 类型进行加速，在本文中我们将使用这些例子。

如果您在普通的 Java 代码中编写一个多行的 `String` 类型，就会最终要使用很多讨厌的 `+` 号，对吗？但是在清单 1 中您可以看到，Groovy 不用再使用这些 `+` 号了，这使您可以编写更加清晰、简单的代码。

清单 **1. Groovy** 中的多行字符串

```
String example1 = "This is a multiline
 string which is going to
 cover a few lines then
 end with a period."
```

Groovy 还支持 here-docs 的概念，如清单 2 所示。*here-doc*是创建格式化 `String`（例如 HTML 和 XML）的一种便利机制。注意 here-doc 语法与普通的 `String` 声明并没有很大的不同，不过它需要类 Python 的三重引号。

清单 **2. Groovy** 中的 **Here-docs**

```
 itext =
"""
 This is another multiline String
 that takes up a few lines. Doesn't
 do anything different from the previous one.
"""
```

Groovy 使用 `GString` 来简化运行时替换。如果您不知道 GString 是什么，我可以确信您之前肯定见过它，而且还可能使用过它。简单来说，`GString` 允许您使用与 *bash* 类似的 `${}` 语法进行替换。`GString` 的优势是您从来都不需要知道自己正在使用的是 `GString` 类型；只需要在 Groovy 中简单地编写 `String` 即可，就仿佛是在 Java 代码中一样。

关于模板引擎

模板引擎已经存在很长一段时间了，几乎在每个现代语言中都可以找到。普通的 Java 语言具有 Velocity 和 FreeMarker；Python 有 Cheetah 和 Ruby ERB；Groovy 也有自己的引擎。要学习更多有关模板引擎的内容，请参阅 参考资料。

清单 **3. Groovy** 中的 **GString**

```
lang = "Groovy"
println "Uncle man, Uncle man, I dig ${lang}."
```

在清单 3 中，我们创建了一个名为 `lang` 的变量，并将其值设置为“Groovy”。我们打印了一个 `GString` 类型的 `String`，并要求将单词 “dig” 后面的内容替换为 `${lang}` 的值。如果实际运行一下，这段代码会打印“Uncle man, Uncle man, I

dig Groovy.” 一切都不错，不是吗？

运行时替换实际上是动态语言的一个通用特性；与其他情况一样，Groovy 还会更进一步。Groovy 的 `GString` 允许您对替换的值调用 *autocall* 方法，当开始构建动态文本时，这会产生很多变化。例如，在清单 4 中，我可以对指定的变量按照 `String` 对象类型调用一个方法（在本例中是 `length()` 方法）。

清单 **4. GString** 自动调用

```
lang = "Groovy"
println "I dig any language with ${lang.length()} characters in i
```

清单 4 中的代码会打印出“I dig any language with 6 characters in its name!”在下一节中，我将向您展示如何使用 Groovy 的自动调用特性在您的模板中启用一些复杂的特性。

Groovy 模板

对模板的使用可以分解为两个主要的任务：首先，创建模板；其次，提供映射代码。使用 Groovy 模板框架创建模板与创建 JSP 非常类似，因为您可以重用 JSP 中见过的语法。创建这些模板的关键在于对那些运行时要替换的变量的定义。例如，在清单 5 中，我们为创建 `GroovyTestCase` 定义了一个模板。

清单 **5.** 一个创建 **GroovyTestCase** 的模板

```
import groovy.util.GroovyTestCase
class <%=test_suite %> extends GroovyTestCase {
 <% for(tc in test_cases) {
    println "\tvoid ${tc}() { } "
 }%>
}
```

清单 5 中的模板就类似于一个 JSP 文件，因为我们使用了 `<%` 和 `<%=` 语法。然而，由于 Groovy 的灵活性很好，因此您并不局限于使用 JSP 语法。您还可以自由使用 Groovy 中杰出的 `GString` ，如清单 6 所示。

清单 **6. GString** 的使用

```
<person>
 <name first="${p.fname}" last="${p.lname}"/>
</person>
```

在清单 6 中，我创建了一个简单的模板，它表示一个定义 `person` 元素集合的 XML 文档。您可以看到这个模板期望一个具有 `fname` 和 `lname` 属性、名为 `p` 的对象。

在 Groovy 中定义模板相当简单，其实应该就是这样简单才对。Groovy 模板并不是火箭科学，它们只不过是一种简化从模型中分离视图过程的手段。下一个步骤是编写运行时的映射代码。

运行时映射

既然已经定义了一个模板，我们就可以通过在已定义的变量和运行时值之间建立映射来使用这个模板了。与 Groovy 中常见的一样，有些看来似乎很复杂的东西实际上是非常简单的。我所需要的是一个 `映射`，其关键字是模板中的变量名，键值是运行时的值。

例如，如果一个简单的模板有一个名为 `favlang` 的变量，我就需要与 `favlang` 键值建立一个 `映射`。这个键值可以是根据我自己的喜好选择的任何脚本语言（在本例中，当然是 Groovy）。

在清单 7 中，我们定义了这样一个简单的模板，在清单 8 中，我将向您展示对应的映射代码。

清单 **7.** 用来展示映射的简单代码

```
My favorite dynamic language is ${favlang}
```

清单 8 显示了一个简单的类，它一共做了 5 件事，其中有 2 件是很重要的。您可以说出它们是什么吗？

清单 **8.** 为一个简单的模板映射值

```
package com.vanward.groovy.tmpl
import groovy.text.Template
import groovy.text.SimpleTemplateEngine
import java.io.File
class SimpleTemplate{
 static void main(args) {
   fle = new File("simple-txt.tmpl")
   binding = ["favlang": "Groovy"]
   engine = new SimpleTemplateEngine()
   template = engine.createTemplate(fle).make(binding)
   println template.toString()
 }
}
```

在清单 8 中为这个简单的模板映射值简单得令人吃惊。

首先，我创建了一个 `File` 实例，它指向模板 `simple-txt.tmpl` 。

然后创建了一个 `binding` 对象；实际上，这就是一个 `映射` 。我将在模板中找到的值 `favlang` 映射到 `String` Groovy 上。这种映射是在 Groovy 中使用模板最重要的步骤，或者说在任何具有模板引擎的语言中都是如此。

接下来，我创建了一个 `SimpleTemplateEngine` 实例，在 Groovy 中，它恰巧就是模板引擎框架的一个具体实现。然后我将模板（ `simple-txt.tmpl` ）和 `binding` 对象传递给这个引擎实例。在清单 8 中，第二个重要的步骤是将模板及其 `binding` 对象绑定在一起，这也是使用模板引擎的关键所在。从内部来说，框架将在从 `binding` 对象中找到的值与对应模板中的名字之间建立映射。

清单 8 中的最后一个步骤是打印进程的输出信息。正如您可以看到的一样，创建一个 `binding` 对象并提供正确的映射是件轻而易举的小事，至少在我们这个简单的例子中是如此。在下一节中，我们将会使用一个更加复杂的例子对 Groovy 模板引擎进行测试。

更复杂的模板

在清单 9 中，我已经创建了一个 `Person` 类来表示在 清单 6中定义的 `person` 元素。

清单 9. Groovy 中的 Person 类

```
class Person{
age
fname
lname
String toString(){
 return "Age: " + age + " First Name: " + fname + " Last Name: " +
}
}
```

在清单 10 中，您可以看到对上面定义的 `Person` 类的实例进行映射的代码。

清单 10. 在 Person 类与模板之间建立映射

```
import java.io.File
import groovy.text.Template
import groovy.text.SimpleTemplateEngine
class TemplatePerson{
 static void main(args) {
   pers1 = new Person(age:12, fname:"Sam", lname:"Covery")
   fle = new File("person_report.tmpl")
   binding = ["p":pers1]
   engine = new SimpleTemplateEngine()
   template = engine.createTemplate(fle).make(binding)
   println template.toString()
 }
}
```

上面的代码看起来很熟悉，不是吗？实际上，它与 清单 8非常类似，不过增加了一行创建 `pers1` 实例的代码。现在，再次快速查看一下 清单 6中的代码。您看到模板是如何引用属性 `fname` 和 `lname` 的了吗？我所做的操作是创建一个 `Person` 实例，其 `fname` 属性设置为“Sam”，属性 `lname` 设置为“Covery”。

在运行清单 10 中的代码时，输出结果是 XML 文件，用来定义 `person` 元素，如清单 11 所示。

清单 **11. Person** 模板的输出结果

```
<person>
 <name first="Sam" last="Covery"/>
</person>
```

映射一个列表

在 清单 5中，我为 `GroovyTestCase` 定义了一个模板。现在如果您看一下这个模板，就会注意到这个定义有一些逻辑用于在一个集合上迭代。在清单 12 中，您将看到一些非常类似的代码，不过这些代码的逻辑是用来映射一个测试用例 `列表` 的。

清单 **12.** 映射测试用例列表

```
fle = new File("unit_test.tmpl")
coll = ["testBinding", "testToString", "testAdd"]
binding = ["test_suite":"TemplateTest", "test_cases":coll]
engine = new SimpleTemplateEngine()
template = engine.createTemplate(fle).make(binding)
println template.toString()
```

查看一下 清单 5，它显示了模板期望一个名为“test_cases”的 `列表` —— 在清单 12 中它定义为 `coll`，包含 3 个元素。我简单地将 `coll` 设置为“test_cases”绑定对象中的键值，现在代码就准备好运行了。

现在应该十分清楚了，Groovy 模板非常容易使用。它还可以促进无所不在的 MVC 模式的使用；更重要的是，它们可以通过表示视图来支持转换为 MVC 代码。在下一节中，我将向您展示如何对上一篇文章中的一个例子应用本文中所学到的知识。

使用模板重构之前的例子

在使用 Groovy 编写 Ant 脚本的专栏中，我曾经编写了一个简单的工具，它对类文件产生校验和报告。如果您还记得，我当时笨拙地使用 `println` 语句来产生 XML 文件。尽管我只能接受这段代码，但它是如此地晦涩，这您只需要看一下清单 13 就会一目了然。

清单 **13.** 糟糕的代码

```
nfile.withPrintWriter{ pwriter |
  pwriter.println("<md5report>")
    for(f in scanner){
       f.eachLine{ line |
         pwriter.println("<md5 class='" + f.path + "' value='" + li
       }
    }
  pwriter.println("</md5report>")
}
```

为了帮您回顾一下有关这段内容的记忆，清单 13 中的代码使用了一些数据，并使用 `PrintWriter` 将其写入一个文件中（ `nfile` 实例）。注意我是如何将报告的视图组件（XML）硬编码在 `println` 中的。这种方法的问题是它不够灵活。之后，如果我需要进行一些修改，就只能进入到 Groovy 脚本的逻辑中进行修改。在更糟糕的情况下，设想一下一个非程序员想要进行一些修改会是什么样子。Groovy 代码会受到很大的威胁。

将该脚本中的视图部分移动到模板中可以使维护更加方便，因为修改模板是任何人都会涉及的一个过程，因此我在此处也会这样做。

定义模板

现在我将开始定义模板了 —— 它看起来更加类似于想要的输出结果，采用了一些逻辑来循环遍历一组类。

清单 **14.** 为原来的代码应用模板

```
<md5report>
<% for(clzz in clazzes) {
 println "<md5 class=\"${clzz.name}\" value=\"${clzz.value}\"/>"
}%>
</md5report>
```

清单 14 中定义的模板与为 `GroovyTestCase` 定义的模板类似，其中包括循环遍历一个集合的逻辑。还要注意我在此处混合使用了 JSP 和 `GString` 的语法。

编写映射代码

定义好模板之后，下一个步骤是编写运行时的映射代码。我需要将原来的写入文件的逻辑替换为下面的代码：构建一个 `ChecksumClass` 对象集合，然后将这些对象放到 `binding` 对象中。

这个模型然后就会变成清单 15 中定义的 `ChecksumClass` 。

清单 **15.** 在 **Groovy** 中定义的 **CheckSumClass**

```
class CheckSumClass{
 name
 value
 String toString(){
  return "name " + name + " value " + value
 }
}
```

Groovy 中类的定义非常简单，不是吗？

创建集合

接下来，我需要重构刚才写入文件的那段代码 —— 这一次采用一定的逻辑使用 `ChecksumClass` 构造一个列表，如清单 16 所示。

清单 **16.** 重构代码创建一个 **ChecksumClass** 的集合

```
clssez = []
for(f in scanner){
 f.eachLine{ line |
  iname = formatClassName(bsedir, f.path)
  clssez << new CheckSumClass(name:iname, value:line)
 }
}
```

清单 16 显示了使用类 Ruby 的语法将对象添加到 `列表` 中是多么简单 —— 这就是 _奇妙的_groovy。我首先使用 `[]` 语法创建 `清单` 。然后使用简短的 `for` 循环，后面是一个带有闭包的迭代器。这个闭包接受每一个 `line` （在本例中是一个校验和值），并创建一个新定义的 `CheckSumClass` 实例（使用 Groovy 的自动生成的构造函数），并将二者添加到集合中。还不错 —— 这样编写起来也很有趣。

添加模板映射

我需要做的最后一件事情是添加模板引擎特定的代码。这段代码将执行运行时映射，并将对应的格式化后的模板写入原始的文件中，如清单 17 所示。

清单 **17.** 使用模板映射重构原来的代码

```
fle = new File("report.tmpl")
binding = ["clazzes": clzzez]
engine = new SimpleTemplateEngine()
template = engine.createTemplate(fle).make(binding)
nfile.withPrintWriter{ pwriter |
 pwriter.println template.toString()
}
```

现在，清单 17 中的代码对您来说太陈旧了。我利用了清单 16 中的 `列表` ，并将其放入 `binding` 对象。然后读取 `nfile` 对象，并将相应的输出内容从 清单 14 中的映射模板写入文件中。

在将这些内容都放入清单 18 之前，您可能希望返回 清单 13最后看一眼开始时使用的那段蹩脚的代码。下面是新的代码，您可以进行比较一下：

清单 **18.** 看，新的代码！

```
/**
*
*/
buildReport(bsedir){
ant = new AntBuilder()
scanner = ant.fileScanner {
  fileset(dir:bsedir) {
    include(name:"**/*class.md5.txt")
  }
}
rdir = bsedir + File.separator + "xml" + File.separator
file = new File(rdir)
if(!file.exists()){
  ant.mkdir(dir:rdir)
}
nfile = new File(rdir + File.separator + "checksum.xml")
clssez = []
for(f in scanner){
  f.eachLine{ line |
   iname = formatClassName(bsedir, f.path)
   clssez << new CheckSumClass(name:iname, value:line)
  }
}
fle = new File("report.tmpl")
binding = ["clazzes": clzzez]
engine = new SimpleTemplateEngine()
template = engine.createTemplate(fle).make(binding)
nfile.withPrintWriter{ pwriter |
  pwriter.println template.toString()
}
}
```

现在，虽然我没有声明要编写非常漂亮的代码，但是这些代码当然不会像原来的代码一样 糟糕了。回顾一下，我所干的事情不过是将一些蹩脚的 `println` 替换成 Groovy 的更精巧的模板代码。（一些熟悉重构的人可能会说我应该使用 *Extract Method*进一步对代码进行优化。）

结束语

在本月的教程中，我希望已经向您展示了 Groovy 的 视图。更 明确地说，当您需要快速开发一些需要视图的简单程序时，Groovy 的模板框架是普通 Java 编码的一种很好的替代品。模板是很好的一种抽象，如果使用正确，它可以极大地降低应用程序的维护成本。

下一个月，我将向您展示如何使用 Groovy 来构建使用 Groovlets 的 Web 应用程序。现在，享受使用 Groovy 的模板开发吧！

下载

描述	名字	大小

j-pg02155-source.zip | 2.18 KB |

实战 Groovy: 用 Groovy 进行 JDBC 编程

用 *GroovySql* 构建下一个报告应用程序

这个月，随着 Andrew Glover 向您演示如何用 GroovySql 构建简单的数据报告应用程序，您对 Groovy 的实用知识会更进一步。GroovySql 结合利用闭包（closure）和迭代器（iterator），把资源管理的负担转移到 Groovy 框架本身，从而简化了 Java 数据库连通性（Java Database Connectivity，JDBC）的编程。

在 实战 *Groovy*系列的前几期中，您已经了解了 Groovy 的一些非常优美的特性。在 第 1 期中，学习了如何用 Groovy 对普通的 Java™ 代码进行更简单、更迅速的单元测试。在 第 2 期中，看到了 Groovy 能够给 Ant 构建带来的表现能力。这一次您会发现 Groovy 的另一个实际应用，即如何用它迅速地构建基于 SQL 的报告应用程序。

脚本语言对于迅速地构建报告应用程序来说是典型的优秀工具，但是构建这类应用程序对于 Groovy 来说特别容易。Groovy 轻量级的语法可以消除 Java 语言的 JDBC 的一些繁冗之处，但是它真正的威力来自闭包，闭包很优雅地把资源管理的责任从客户机转移到框架本身，因此更容易举重若轻。

在本月的文章中，我先从 GroovySql 的概述开始，然后通过构建一个简单的数据报告应用程序，向您演示如何将它们投入工作。为了从讨论中得到最大收获，您应当熟悉 Java 平台上的 JDBC 编程。您可能还想回顾 上个月对 Groovy 中闭包的介绍，因为它们在这里扮演了重要的角色。但是，本月关注的重点概念是迭代（iteration），因为迭代器在 Groovy 对 JDBC 的增强中扮演着重要角色。所以，我将从 Groovy 中迭代器的概述开始。

进入迭代器

迭代是各种编程环境中最常见、最有用的技术。 迭代器是某种代码助手，可以让您迅速地访问任何集合或容器中的数据，每次一个数据。Groovy 把迭代器变成隐含的，使用起来更简单，从而改善了 Java 语言的迭代器概念。在清单 1 中，您可以看到使用 Java 语言打印 `String` 集合的每个元素需要做的工作。

清单 **1.** 普通 **Java** 代码中的迭代器

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class JavaIteratorExample {
 public static void main(String[] args) {
    Collection coll = new ArrayList();
    coll.add("JMS");
    coll.add("EJB");
    coll.add("JMX");
    for(Iterator iter = coll.iterator(); iter.hasNext();){
       System.out.println(iter.next());
    }
 }
}
```

在清单 2 中，您可以看到 Groovy 如何简化了我的工作。在这里，我跳过了 `Iterator` 接口，直接在集合上使用类似迭代器的方法。而且， Groovy 的迭代器方法接受闭包，每个迭代中都会调用闭包。清单 2 显示了前面基于 Java 语言的示例用 Groovy 转换后的样子。

清单 **2. Groovy** 中的迭代器

```
class IteratorExample1{
  static void main(args) {
    coll = ["JMS", "EJB", "JMX"]
    coll.each{ item | println item }
  }
}
```

您可以看到，与典型的 Java 代码不同，Groovy 在允许我传递进我需要的行为的同时，控制了特定于迭代的代码。使用这个控制，Groovy 干净漂亮地把资源管理的责任从我手上转移到它自己身上。让 Groovy 负责资源管理是极为强大的。它还使编程工作更加容易，从而也就更迅速。

关于本系列

把任何一个工具集成进您的开发实践的关键是，知道什么时候使用它而什么时候把它留在箱子中。脚本语言可以是您的工具箱中极为强大的附件，但是只有在恰当地应用到适当的场景时才是这样。为了这个目标， *实战 Groovy*这一系列文章专门介绍了 Groovy 的实际应用，并教给您什么时候、如何成功地应用它们。

GroovySql 简介

Groovy 的 SQL 魔力在于一个叫做 GroovySql 的精致的 API。使用闭包和迭代器，GroovySql 干净漂亮地把 JDBC 的资源管理职责从开发人员转移到 Groovy 框架。这么做之后，就消除了 JDBC 编程的繁琐，从而使您可以把注意力放在查询和查询结果上。

如果您忘记了普通的 Java JDBC 编程有多麻烦，我会很高兴提醒您！在清单 3 中，您可以看到用 Java 语言进行的一个简单的 JDBC 编程示例。

清单 **3.** 普通 **Java** 的 **JDBC** 编程

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class JDBCExample1 {
 public static void main(String[] args) {
   Connection con = null;
   Statement stmt = null;
   ResultSet rs = null;
   try{
     Class.forName("org.gjt.mm.mysql.Driver");
     con = DriverManager.getConnection("jdbc:mysql://localhost:33(
          "words", "words");
     stmt = con.createStatement();
     rs = stmt.executeQuery("select * from word");
     while (rs.next()) {
       System.out.println("word id: " + rs.getLong(1) +
           " spelling: " + rs.getString(2) +
           " part of speech: " + rs.getString(3));
     }
   }catch(SQLException e){
     e.printStackTrace();
   }catch(ClassNotFoundException e){
     e.printStackTrace();
   }finally{
     try{rs.close();}catch(Exception e){}
     try{stmt.close();}catch(Exception e){}
     try{con.close();}catch(Exception e){}
  }
 }
}
```

哇。清单 3 包含近 40 行代码，就是为了查看表中的内容！如果用 GroovySql，您猜要用多少行？如果您猜超过 10 行，那么您就错了。请看清单 4 中，Groovy 替我处理底层资源，从而非常漂亮地让我把注意力集中在手边的任务 —— 执行一个简单的查询。

清单 **4.** 欢迎使用 **GroovySql** ！

```
import groovy.sql.Sql
class GroovySqlExample1{
 static void main(args) {
   sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "wo
          "words", "org.gjt.mm.mysql.Driver")
   sql.eachRow("select * from word"){ row |
      println row.word_id + " " + row.spelling + " " + row.part_o
   }
 }
}
```

真不错。只用了几行，我就编码出与清单 3 相同的行为，不用关闭 `Connection`，也不用关闭 `ResultSet`，或者在 JDBC 编程中可以找到的任何其他熟悉的重要特性。这是多么激动人心的事情啊，并且还是如此容易。现在让我来仔细介绍我是如何做到的。

执行简单的查询

在 清单 4的第一行中，我创建了 Groovy 的 `Sql` 类的实例，用它来连接指定的数据库。在这个例子中，我创建了 `Sql` 实例，指向在我机器上运行的 MySQL 数据库。到现在为止都非常基本，对么？真正重要的是一下部分，迭代器和闭包一两下就显示出了它们的威力。

请把 `eachRow` 方法当成传进来的查询生成的结果上的迭代器。在底层，您可以看到返回了 JDBC `ResultSet` 对象，它的内容被传递进 `for` 循环。所以，每个迭代都要执行我传递进去的闭包。如果在数据库中找到的 `word` 表只有三行，那么闭包就会执行三次 —— 打印出 `word_id` 、 `spelling` 和 `part_of_speech` 的值。

如果将等式中我指定的变量 `row` 去掉，而使用 Groovy 的一个隐含变量 `it` （它恰好就是迭代器的实例），代码可以进一步简化。如果我这样做，那么前面的代码就可以写成清单 5 所示的这样：

清单 5. GroovySql 中的 `it` 变量

```
import groovy.sql.Sql
class GroovySqlExample1{
 static void main(args) {
   sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "wo
          "words", "org.gjt.mm.mysql.Driver")
   sql.eachRow("select * from word"){ println it.spelling +  " ${
 }
}
```

在这个代码中，我可以删除 `row` 变量，用 `it` 代替。而且，我还能在 `String` 语句中引用 `it` 变量，就像我在 `${it.part_of_speech}` 中所做的那样。

执行更复杂的查询

前面的例子相当简单，但是 GroovySql 在处理更复杂的数据操纵查询（例如 `insert` 、 `update` 和 `delete` 查询）时，也是非常可靠的。 对于这些查询，您没有必要用迭代器，所以 Groovy 的 `Sql` 对象另外提供了 `execute` 和 `executeUpdate` 方法。这些方法让人想起普通的 JDBC `statement` 类，它也有 `execute` 和 `executeUpdate` 方法。

在清单 6 中，您看到一个简单的 `insert` ，它再次以 `${}` 语法使用变量替换。这个代码只是向 `word` 表插入一个新行。

清单 **6.** 用 **GroovySql** 进行插入

```
wid = 999
spelling = "Nefarious"
pospeech = "Adjective"
sql.execute("insert into word (word_id, spelling, part_of_speech)
  values (${wid}, ${spelling}, ${pospeech})")
```

Groovy 还提供 `execute` 方法的一个重载版本，它接收一列值，这些值与查询中发现的 `?` 元素对应。在清单 7 中，我简单地查询了 `word` 表中的某个行。在底层，GroovySql 创建了普通 Java 语言 `java.sql.PreparedStatement` 的一个实例。

清单 **7.** 用 **GroovySql** 创建 **PreparedStatement** 的实例

```
val = sql.execute("select * from word where word_id = ?", [5])
```

更新的方式基本相同，也使用 `executeUpdate` 方法。还请注意，在清单 8 中 `executeUpdate` 方法接收一列值，与查询中的 `?` 元素对应。

清单 **8.** 用 **GroovySql** 进行更新

```
nid = 5
spelling = "Nefarious"
sql.executeUpdate("update word set word_id = ? where spelling = ?'
```

删除实际上与插入相同，当然，语法不同，如清单 9 所示。

清单 **9.** 用 **GroovySql** 进行删除

```
sql.execute("delete from word where word_id = ?" , [5])
```

简化数据操纵

任何想简化 JDBC 编程的 API 或工具最好有一些好的数据操纵特性，在这一节中，我要向您再介绍三个。

数据集 **(DataSet)**

构建于 GroovySql 简单性的基础之上，GroovySql 支持 `DataSet` 类型的概念，这基本上是数据库表的对象表示。使用 `DataSet` ，您可以在行中遍历，也可以添加新行。实际上，用数据集是方便地表示表格的公共数据集合的方式。

但是，目前 GroovySql `DataSet` 类型的不足之处是它们没有代表关系；它们只是与数据库表的一对一映射。在清单 10 中，我创建了一个来自 `word` 表的 `DataSet` 。

清单 **10.** 用 **GroovySql** 创建数据集

```
import groovy.sql.Sql
class GroovyDatasetsExample1{
 static void main(args) {
   sql = Sql.newInstance("jdbc:mysql://localhost:3306/words", "wor
         "words", "org.gjt.mm.mysql.Driver")
   words = sql.dataSet("word")
   words.each{ word |
    println word.word_id + " " + word.spelling
   }
   words.add(word_id:"9999", spelling:"clerisy", part_of_speech:"N
 }
}
```

您可以看到，GroovySql 的 `DataSet` 类型可以容易地用 `each` 方法对表的内容进行遍历，容易地用 `add` 方法添加新行， `add` 方法接受一个 `map` 表示需要的数据。

使用存储过程和负索引

存储过程调用和负索引（negative indexing）可能是数据操纵的重要方面。GroovySql 使存储过程调用简单得就像在 `Sql` 类上使用 `call` 方法一样。对于负索引，GroovySql 提供了自己增强的 `ResultSet` 类型，它工作起来非常像 Groovy 中的 *collections*。例如，如果您想获取结果集中的最后一个项目，您可以像清单 11 所示的那样做：

清单 **11.** 用 **GroovySql** 进行负索引

```
sql.eachRow("select * from word"){ grs |
  println "-1  = " + grs.getAt(-1) //prints spelling
  println "2  = " + grs.getAt(2) //prints spelling
}
```

您在清单 11 中可以看到，提取结果集的最后一个元素非常容易，只要用 -1 作索引就可以。如果想试试，我也可以用索引 2 访问同一元素。

这些例子非常简单，但是它们能够让您很好地感觉到 GroovySql 的威力。我现在要用一个演示目前讨论的所有特性的实际例子来结束本月的课程。

编写一个简单的报告应用程序

报告应用程序通常要从数据库拖出信息。在典型的业务环境中，可能会要求您编写一个报告应用程序，通知销售团队当前的 Web 销售情况，或者让开发团队对系统某些方面（例如系统的数据库）的性能进行日常检测。

为了继续这个简单的例子，假设您刚刚部署了一个企业范围的 Web 应用程序。当然，因为您在编写代码时还（用 Groovy）编写了充足的单元测试，所以它运行得毫无问题；但是您还是需要生成有关数据库状态的报告，以便调优。您想知道客户是如何使用应用程序的，这样才能发现性能问题并解决问题。

通常，时间约束限制了您在这类应用程序中能够使用的提示信息的数量。但是您新得到的 GroovySql 知识可以让您轻而易举地完成这个应用程序，从而有时间添加更多您想要的特性。

细节

在这个例子中，您的目标数据库是 MySQL，它恰好支持用查询发现状态信息这一概念。以下是您有兴趣的状态信息：

- 运行时间。
- 处理的全部查询数量。
- 特定查询的比例，例如 `insert` 、 `update` 和 `select` 。

用 GroovySql 从 MySQL 数据库得到这个信息太容易了。由于您正在为开发团队构建状态信息，所以您可能只是从一个简单的命令行报告开始，但是您可以在后面的迭代中把报告放在 Web 上。这个报告例子的用例看起来可能像这样：

1.	连接到我们的应用程序的活动数据库。
2.	发布 `show status` 查询并捕获：
	a. 运行时间
	b. 全部查询数
	c. 全部 `insert` 数
	d. 全部 `update` 数
	e. 全部 `select` 数
3.	使用这些数据点，计算：
	a. 每分钟查询数
	b. 全部 `insert` 查询百分比
	c. 全部 `update` 查询百分比
	d. 全部 `select` 查询百分比

在清单 12 中，您可以看到最终结果：一个将会报告所需数据库统计信息的应用程序。代码开始的几行获得到生产数据库的连接，接着是一系列 `show status` 查询，让您计算每分钟的查询数，并按类型把它们分开。请注意像 `uptime` 这样的变量如何在定义的时候就创建。

清单 **12.** 用 **GroovySql** 进行数据库状态报告

```
import groovy.sql.Sql
class DBStatusReport{
  static void main(args) {
     sql = Sql.newInstance("jdbc:mysql://yourserver.anywhere/tiger'
        "tiger", "org.gjt.mm.mysql.Driver")
     sql.eachRow("show status"){ status |
        if(status.variable_name == "Uptime"){
           uptime =  status[1]
        }else if (status.variable_name == "Questions"){
           questions =  status[1]
        }
     }
     println "Uptime for Database: " + uptime
     println "Number of Queries: " + questions
     println "Queries per Minute = "
         + Integer.valueOf(questions)/Integer.valueOf(uptime)
     sql.eachRow("show status like 'Com_%'"){ status |
        if(status.variable_name == "Com_insert"){
           insertnum =  Integer.valueOf(status[1])
        }else if (status.variable_name == "Com_select"){
           selectnum =  Integer.valueOf(status[1])
        }else if (status.variable_name == "Com_update"){
           updatenum =  Integer.valueOf(status[1])
       }
    }
    println "% Queries Inserts = " + 100 * (insertnum / Integer.va]
    println "% Queries Selects = " + 100 * (selectnum / Integer.va]
    println "% Queries Updates = " + 100 * (updatenum / Integer.va]
    }
}
```

结束语

在 实战 *Groovy*本月的这一期文章中，您看到了 GroovySql 如何简化 JDBC 编程。这个干净漂亮的 API 把闭包和迭代器与 Groovy 轻松的语法结合在一起，有助于在 Java 平台上进行快速数据库应用程序开发。最强大的是，GroovySql 把资源管理任务从开发人员转移到底层的 Groovy 框架，这使您可以把精力集中在更加重要的查询和查询结果上。但是不要只记住我这句话。下次如果您被要求处理 JDBC 的麻烦事，那时可以试试小小的 GroovySql 魔力。然后给我发封电子邮件告诉我您的体会。

在 实战 *Groovy*下一月的文章中，我要介绍 Groovy 模板框架的细节。您会发现，用这个更加聪明的框架创建应用程序的视图组件简直就是小菜一碟。

实战 Groovy: 用 Groovy 进行 Ant 脚本编程

为更具表现力、更可控的构建而组合使用 *Ant* 和 *Groovy*

Ant 和 Maven 两者在构建处理工具的世界中占统治地位。但是 XML 却凑巧是一种非常没有表现力的配置格式。在“实战 Groovy”这个新系列的第 2 期中，Andrew Glover 将介绍 Groovy 的生成器实用工具，这个工具能够极其容易地把 Groovy 与 Ant 和 Maven 结合在一起，形成更具表现力、更可控的构建。

Ant 作为 Java 项目构建工具的普遍性和实用性是无法超越的。即使是 Maven 这个构建领域的新锐工具，也要把自己的许多强大能力归功于从 Ant 学到的经验。但是，这两个工具有共同的不足之处：扩展性。即使 XML 的可移植性在促进 Ant 和 Maven 走向开发前端上扮演了主要角色，XML 作为构建配置格式的角色仍然或多或少地限制了构建过程的表现力。

例如，虽然在 Ant 和 Maven 中都有条件逻辑，但是用 XML 表示有些繁琐。而且，虽然可以定义定制任务来扩展 Ant 的构建过程，但是这么做通常会把应用程序的行为局限在有限的沙箱中。因此，在本文中，我将向您演示如何把 Groovy 和 Ant 在 Maven 内部 结合起来，形成更强大的表现力，从而对构建过程进行更好的行为控制。

很快您自己就会看到，Groovy 给 Ant 和 Maven 都带来了令人瞩目的增强，Groovy 的优点在于它常常能恰到好处地弥补 XML 的遗漏！实际上，看起来 Groovy 的创建者肯定也曾经历过用 XML 实施 Ant 和 Maven 构建过程的痛苦，因为他们引入了 `AntBuilder`，这是一个强大的新工具，支持在 Groovy 脚本中利用 Ant。

在 实战 *Groovy* 的这一期中，我将向您展示不用 XML，转用 Groovy 作为您的构建配置格式，对构造过程进行增强是多么容易。闭包（closure）是 Groovy 的一个重要特性，而且是这门语言之所以与众不同的核心，所以我在转到下一节之前，先要对闭包做一个快速回顾。

快速回顾闭包

就像 Groovy 自己的一些非常著名的祖先一样，Groovy 也支持 匿名函数 或 闭包（*closure*） 的概念。如果您曾经用 Python 或 Jython 编写过代码，那么您可能对闭包比较熟悉，在这两种语言中，都是用 `lambda` 关键字引入闭包。在 Ruby 中，如果 没有 利用块或闭包，那么您就不得不实际地编写脚本。即使是 Java 语言，也要通过它的 匿名内部类，对匿名函数提供了有限形式的支持。

在 Groovy 中，闭包是能够封装行为的第一级匿名函数。当 Ruby 的缔造者 Yukihiro Matsumoto 注意到这些强大的第一类对象可以“传递给另一个函数，然后函数可以调用传入的 [闭包]”时，也开始染指闭包的应用（请参阅 参考资料，查看完整的采访）。当然，亲自操作 Groovy 是了解闭包对于这门美好的令人激动的语言是一笔多么大的财富的最好方法。

关于这一系列的教程

把任何一个工具集成进您的开发实践的关键是，知道什么时候使用它而什么时候把它留在箱子中。脚本语言可以是您的工具箱中极为强大的附件，但是只有在恰当地应用到适当地场景中才是这样。为了这个目标， 实战 *Groovy* 是一系列文章，专门介绍 Groovy 的实际应用，并教给您什么时候、如何成功地应用它们。

闭包实例

在清单 1 中，我又使用了一些本系列的第 1 篇文章曾经用过的代码；但是这一次的重点是闭包。如果您已经读过那篇文章（请参阅 参考资料），那么您可以回忆起我用来演示用 Groovy 进行单元测试的基于 Java 的包过滤对象。这次，我还用相同的示例开始，但是用闭包对它进行了极大的增强。下面是基于 Java 的 `Filter` 接口。

清单 **1.** 还记得这个简单的 **Java Filter** 接口吗？

```
public interface Filter {
  void setFilter(String fltr);
  boolean applyFilter(String value);
}
```

上次，在定义了 `Filter` 类型之后，我接着定义了两个实现，这两个实现名为 `RegexPackageFilter` 和 `SimplePackageFilter` ，它们分别使用了正则表达式和简单的 `String` 操作。

对于没用闭包写成的代码来说，到现在为止还算不错。在清单 2 中，您会开始看到只有一点语法上的变化，代码就不同了（更好了！）。我将通过定义通用的 `Filter` 类型（如下所示）开始，但是这次是用 Groovy 定义的。请注意与 `Filter` 类关联的 `strategy` 属性。这个属性是闭包的一个实例，在执行 `applyFilter` 方法的时候可以调用它。

清单 **2.** 更加 **Groovy** 的过滤器 **——** 使用闭包

```
class Filter{
   strategy
   boolean applyFilter(str){
      return strategy.call(str)
   }
}
```

添加闭包意味着我不用像在原来的 `Filter` 接口中那样必须定义一个接口类型，或者依赖特定的实现才能得到期望的行为。相反，我可以定义一个通用的 `Filter` 类型，并给它提供一个闭包，让它在执行 `applyFilter` 方法期间应用。

下一步是定义两个闭包。第一个通过对指定参数应用简单的 `String` 操作来模拟 `SimplePackageFilter`（来自前一篇文章）。当创建新的 `Filter` 类型时，对应的名为 `simplefilter` 的闭包会被传递到构造器中。第二个闭包（在几个进行代码验证的 `assert` 之后出现）对指定的 `String` 应用正则表达式。这次我还是要创建一个新的 `Filter` 类型，向正则表达式传递一个名为 `rfilter` 的闭包，并执行一些 `assert`，以确保每件事都正常。所有细节如清单 3 所示:

清单 **3.** 使用 **Groovy** 闭包的简单魔术

```
simplefilter = { str |
   if(str.indexOf("java.") >= 0){
     return true
   }else{
     return false
   }
}

fltr = new Filter(strategy:simplefilter)
assert !fltr.apply("test")
assert fltr.apply("java.lang.String")

rfilter = { istr |
   if(istr =~ "com.vanward.*"){
     return true
   }else{
     return false
   }
}

rfltr = new Filter(strategy:rfilter)
assert !rfltr.apply("java.lang.String")
assert rfltr.apply("com.vanward.sedona.package")
```

非常令人难忘，对吧？使用闭包，我能够把对期望行为的定义推迟到运行时 —— 所以不必像在以前的设计中要做的那样再定义新的 `Filter` 类型并编译它。虽然我用 Java 代码时能用匿名内部类做类似的事情，但是用 Groovy 更容易、神秘性更少一些。

闭包确实是非常强大的家伙。它们还代表处理行为的另外一种方式 —— 而 Groovy（以及它的远亲 Ruby）对它有非常严重的依赖。所以闭包会是我们下一个主题的要点，下一个主题是用生成器进行构建。

用生成器进行构建

使 Groovy 中的 Ant 更迷人的核心之处是 生成器。实际上，生成器允许您很方便地在 Groovy 中表示树形数据结构，例如 XML 文档。而且，女士们先生们请看，秘密在这：使用生成器，特别是 `AntBuilder` ，您可以毫不费力地构造 Ant 的 XML 构建文件，不必处理 XML 就可以 执行生成的行为。而这并不是在 Groovy 中使用 Ant 的惟一优点。与 XML 不同，Groovy 是非常有表现力的开发环境，在这个环境中，您可以容易地编写循环结构、条件选择代码，甚至可以利用“重用”的威力，而不必像以前那样，费力地用剪切-粘贴操作来创建新 build.xml 文件。而且您做这些工作时，完全是在 Java 平台中！

生成器的优点，尤其是 Groovy 的 `AntBuilder` ，在于它们的语法表示完全体现了它们所代表的 XML 文件的逻辑进程。被附加在 `AntBuilder` 实例上的方法与对应的 Ant 任务匹配；同样的，这些方法可以接收参数（以 `map` 的形式），参数对应着任务的属性。而且，嵌套标签（例如 `include` 和 `fileset` ）也定义成闭包。

构建块：示例 1

我要用一个超级简单的示例向您介绍生成器：一个叫做 `echo` 的 Ant 任务。在清单 4 中，我创建了一个普通的、每天都会用到的 Ant 的 `echo` 标记的 XML 版本（用在这不要奇怪）：

清单 **4. Ant** 的 **Echo** 任务

```
<echo message="This was set via the message attribute"/>
<echo>Hello World!</echo>
```

事情在清单 5 中变得更有意思了：我用相同的 Ant 标签，并在 Groovy 中用 `AntBuilder` 类型重新定义了它。注意，我可以使用 `echo` 的属性 `message` ，也可以只传递一个期望的 `String` 。

清单 **5.** 用 **Groovy** 表示的 **Ant** 的 **Echo** 任务

```
ant = new AntBuilder()
ant.echo(message:"mapping it via attribute!")
ant.echo("Hello World!")
```

生成器特别吸引人的地方是它可以让我把普通的 Groovy 特性与生成器语法混合，从而创建丰富的行为集。在清单 6 中，您应当开始看出可能性是无穷的：

清单 **6.** 用 **Groovy** 和 **Ant** 进行流控制（**flow control**）

```
ant = new AntBuilder()
ant.mkdir(dir:"/dev/projects/ighr/binaries/")
try{
    ant.javac(srcdir:"/dev/projects/ighr/src",
       destdir:"/dev/projects/ighr/binaries/" )
}catch(Throwable thr){
    ant.mail(mailhost:"mail.anywhere.com", subject:"build failure"
       from(address:"buildmaster@anywhere.com", name:"buildmaster"
       to(address:"dev-team@anywhere.com", name:"Development Team"
       message("Unable to compile ighr's source.")
    }
}
```

在这个示例中，我要捕获源代码编译时的错误条件。注意 `catch` 块中定义的 `mail` 对象如何接受定义了 `from` 、 `to` 和 `message` 属性的闭包。

_哎呀！_Groovy 的功能真多！当然，知道什么时候应用这么聪明的特性是问题的关键，而我们都在不断地为之努力。幸运的是，实践出真知，一旦您开始使用 Groovy（或者为了这个原因使用任何脚本语言），您就会找到许多在实际工作中使用它的机会；从中了解它在哪里才真正适用。在下一节中，我将查看一个典型的、现实的示例，并在其中使用一些在这里介绍的特性。

应用 Groovy

对于这个示例，我们假设需要为我的代码定期建立校验和（checksum）报告。一旦实现了这个报告，就可以用它在我需要的时候检验文件的完整性。下面是校验和报告的高级技术用例：

1. 编译所有的源代码。
2. 对二进制类文件运行 md5 算法。
3. 创建简单的报告，列出每个类文件及其对应的校验和。

完全摈弃 Ant 或 Maven，整个构建过程都使用 Groovy，在这个例子中，这样做有点极端。但实际上，正如我前面解释的，Groovy 是这些工具的极大 增强，而 不是替代。所以，用 Groovy 的表现力处理后两项任务，而把第一步信托给 Ant 或 Maven，这样做才有意义。

实际上，我们只要假设我在第一步中用的是 Maven，因为坦白地说，它是我个人偏爱的构建平台。在 Maven 中可以很容易地用 `java:compile` 和 `test:compile` 目标编译源文件；所以我保持这部分内容原封不动，让我的新目标引用前面的目标，把前面的目标作为前提条件。实际就是这样 —— 使用编译好的源文件，我准备继续运行校验和工具（checksum utility）；但是首先还需要做一些简单的设置。

设置 Md5ReportBuilder

为了通过 Ant 运行这个漂亮的校验和工具，我需要两条信息：哪个目录包含要进行校验和处理的文件，报告要写到哪个目录。对于工具的第一个参数，我希望它是一个用逗号分隔的目录列表。对后一个参数，我希望是报告目录。

我决定调用工作类 `Md5ReportBuilder` 。清单 7 中定义了它的 `main` 方法：

清单 7. Md5ReportBuilder 的 Main 方法

```
static void main(args) {

  assert args[0] && args[1] != null

  dirs = args[0].split(",")
  todir = args[1]

  report = new Md5ReportBuilder()
  report.runCheckSum(dirs)
  report.buildReport(todir)
}
```

上述步骤中的第一步是检查这两个参数是不是传递到工具中。然后我把第一个参数用逗号进行分割，创建了一个数组，保存要在上面运行 Ant 的 `checksum` 任务的目录。最后，我创建 `Md5ReportBuilder` 类的新实例，调用两个方法处理所需要的功能。

添加校验和

Ant 包含一个 `checksum` 任务，调用起来非常容易，但需要传递一个 `fileset` 给它，其中包含目标文件的集合。在这个例子中，目标文件是包含编译后的源文件的目录，以及对应的单元测试文件。我可以通过使用 `for` 循环中的迭代得到这些文件，在这个例子中，是在目录集合中进行迭代。对于每个目录，调用 `checksum` 任务；而且 `checksum` 任务只在 .class 文件上运行，如清单 8 所示：

清单 8. runCheckSum 方法

```
/**
 * runs checksum task for each dir in collection passed in
 */
runCheckSum(dirs){
  ant = new AntBuilder()
  for(idir in dirs){
    ant.checksum(fileext:".md5.txt" ){
      fileset(dir:idir) {
        include(name:"**/*.class")
      }
    }
  }
}
```

构建报告

从这一点起，构建报告就变成了循环练习。读取每个新生成的校验和文件，把该文件对应的信息送到 `PrintWriter` 方法，然后将 XML 写入文件 —— 虽然用的是最丑陋的形式，如清单 9 所示：

清单 **9.** 构建报告

```
buildReport(bsedir){
  ant = new AntBuilder()
  scanner = ant.fileScanner {
    fileset(dir:bsedir) {
      include(name:"**/*class.md5.txt")
    }
  }
  rdir = bsedir + File.separator + "xml" + File.separator
  file = new File(rdir)
  if(!file.exists()){
    ant.mkdir(dir:rdir)
  }
  nfile = new File(rdir + File.separator + "checksum.xml")
  nfile.withPrintWriter{ pwriter |
     pwriter.println("<md5report>")
     for(f in scanner){
       f.eachLine{ line |
         pwriter.println("<md5 class='" + f.path + "' value='" + li
       }
     }
     pwriter.println("</md5report>")
  }
}
```

那么，如何处理这个报告呢？首先，我用 `FileScanner` 找到清单 8 中的 `checksum` 方法创建的每个校验和文件。然后，我创建一个新目录，在这个目录中再创建一个新文件。（如果我能通过一个简单的 `if` 语言检查目录是否已经存在，会不会更好一些？）然后我打开对应的文件，并用一个漂亮的闭包从 `scanner` 集合读取每个对应的 `File` 。示例最后用写入 XML 元素的报告内容进行总结。

Goal 就是您的目标！

不熟悉 Maven 的读者可能想知道 `goal` 谈的都是什么。把 Maven 中的 `goal` 当成 Ant 中的 `target` 就可以了。 *goal* 就是组织行为的一种方式。Maven 的 `goal` 被赋予名称，可以通过 `maven` 命令调用它们。在调用时，在指定 `goal` 找到的任务都会被执行。

我敢打赌您立刻就注意到在 `File` 的那些实例上的 `withPrintWriter` 方法是多么强大。我不需要考虑异常或关闭文件，因为它替我处理了每件事。我只是把我希望的行为通过闭包传递给它，然后就准备就绪了！

运行工具

这个 Groovy 巡演中的下一站是把它装配进构建过程，特别是放在我的 maven.xml 文件中。非常幸运的是，这是这个相对容易的练习中最容易的部分，如清单 10 所示：

清单 10. 在 Maven 中运行 Md5ReportBuilder

```
<goal name="gmd5:run" prereqs="java:compile,test:compile">
  <path id="groovy.classpath">
    <ant:pathelement path="${plugin.getDependencyClasspath()}"/>
    <ant:pathelement location="${plugin.dir}"/>
    <ant:pathelement location="${plugin.resources}"/>
  </path>
  <java classname="groovy.lang.GroovyShell" fork="yes">
    <classpath refid="groovy.classpath"/>
    <arg value="${plugin.dir}/src/groovy/com/vanward/md5builder/Md5
    <arg value="${maven.test.dest},${maven.build.dest}"/>
    <arg value="${maven.build.dir}/md5-report"/>
  </java>
</goal>
```

正如前面所解释过的，我已经规定必须在完整的编译之前运行校验和工具，所以，我的目标有两个前提条件： `java:compile` 和 `test:compile` 。类路径 Classpath 总是很重要，所以我专门为了让 Groovy 运行特别创建了一个正确的

classpath。最后清单 10 所示的目标调用 Groovy 的外壳，把要运行的脚本和脚本对应的两个参数传给外壳 —— 第一个是要在其上运行校验和（用逗号分隔的）目录，第二个是报告要写入的目录。

最后一步

完成对 maven.xml 文件的编码之后，就差不多完成了所有要做的事，但是还有最后一步：我需要用所需的依赖关系来更新 `project.xml` 文件。没什么好奇怪的，Maven 需要具有一些必要的 Groovy 依赖项才能工作。这些依赖项是汇编代码、字节码操纵库、公共命令行（commons-cli-处理命令行解析）Ant、以及对应的 ant 启动器。这个示例的依赖项如清单 11 所示：

清单 **11. Groovy** 必需的依赖项

```
<dependencies>
  <dependency>
    <groupId>groovy</groupId>
    <id>groovy</id>
    <version>1.0-beta-6</version>
  </dependency>
  <dependency>
    <groupId>asm</groupId>
    <id>asm</id>
    <version>1.4.1</version>
  </dependency>
  <dependency>
    <id>commons-cli</id>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant-launcher</artifactId>
    <version>1.6.1</version>
  </dependency>
</dependencies>
```

结束语

在 实战 *Groovy* 的第 2 期中，您看到了 Groovy 的表现力和灵活性与 Ant 和 Maven 无与伦比的应用结合在一起时发生了什么。对于任何一个工具，Groovy 都提供了有吸引力的替代 XML 的构建格式。它让您用循环构造和条件逻辑控制程序流，极大地增强了构建过程。

在向您展示 Groovy 的实用一面的同时，这个系列还专门介绍了它最适当的应用。应用任何技术的要点之一是认真思考它要应用的环境。在这个案例中，我向您展示了如何用 Groovy 增强 而不是 替换 已经非常强大的工具：Ant。

下个月，我要介绍 Groovy 的另外一项依赖闭包的特性。GroovySql 是个超级方便的小工具，可以使数据库查询、更新、插入以及所有对应的逻辑管理起来特别容易！下期再见！

实战 Groovy: 用 Groovy 更迅速地对 Java 代码进行单元测试

null

不久以前，developerWorks 的作者 Andrew Glover 撰写了一篇介绍 Groovy 的文章，该文章是 *alt.lang.jre* 系列的一部分，而 Groovy 是一个新提议的用于 Java 平台的标准语言。读者对这篇文章的反应非常热烈，所以我们决定开办这个专栏，提供使用这项热门新技术的实用指导。本文是第一期，将介绍使用 Groovy 和 JUnit 对 Java 代码进行单元测试的一个简单策略。

开始之前，我首先要招认：我是一个单元测试狂。实际上，我总是无法编写足够的单元测试。如果我相当长一段时间都在进行开发，而 没有编写相应的单元测试，我就会觉得紧张。单元测试给我信心，让我相信我的代码能够工作，而且我只要看一下，可以修改它，就不会害怕它会崩溃。

而且，作为一个单元测试狂，我喜欢编写多余的测试用例。但是，我的兴奋不是来自 编写测试用例，而是 看着它们生效。所以，如果我能用更快的方式编写测试，我就能更迅速地看到它们的结果。这让我感觉更好。更快一些！

后来，我找到了 Groovy，它满足了我的单元测试狂，而且至今为止，对我仍然有效。这种新语言给单元测试带来的灵活性，非常令人兴奋，值得认真研究。本文是介绍 Groovy 实践方面的新系列的第一部分，在文中，我将向您介绍使用 Groovy 进行单元测试的快乐。我从概述开始，概述 Groovy 对 Java 平台上的开发所做的独特贡献，然后转而讨论使用 Groovy 和 JUnit 进行单元测试的细节，其中重点放在 Groovy 对 JUnit 的 `TestCase` 类的扩展上。最后，我用一个实用的示例进行总结，用第一手材料向您展示如何把 *groovy* 的这些特性与 Eclipse 和 Maven 集成在一起。

不要再坚持 Java 纯粹主义了！

在我开始介绍用 Groovy 进行单元测试的实际经验之前，我认为先谈谈一个更具一般性的问题 —— 它在您的开发工具箱中的位置，这非常重要。事实是，Groovy 不仅是运行在 Java 运行时环境（JRE）中的脚本语言，它还被提议作为用于 Java 平台的标准语言。正如您们之中的人已经从 *alt.lang.jre* 系列（请参阅 参考资料）中学习到的，在为 Java 平台进行脚本编程的时候，有无数的选择，其中大多数是面向快速应用程序开发的高度灵活的环境。

虽然有这么丰富的选择，但还是有许多开发人选择坚持自己喜欢的、最熟悉的范式：Java 语言。虽然大多数情况下，Java 编程都是很好的选择，但是它有一个非常重要的缺点，蒙住了只看见 Java 的好处的这些人的眼睛。正如一个智者曾经指出的： 如果您仅有的一个工具是一把锤子，那么您看每个问题时都会觉得它像是钉子。我认为这句谚语道出了适用于软件开发的许多事实。

虽然我希望用这个系列说服您 Java 不是也不应当是开发应用程序的惟一选择，但该脚本确实既有适用的地方也有不适用的地方。专家和新手的区别在于：知道什么时候 运用该脚本，什么时候 避免使用它。

关于本系列

把工具整合到开发实践中的关键是了解什么时候使用它，以及什么时候把它留在工具箱中。脚本语言能够成为工具包中极为强大的附件，但是只有正确地应用在适当的场合时才是这样。为了实现 实战 *Groovy* 系列文章这个目标，我专门研究了 Groovy 的一些实战，教给您什么时候怎样才能成功地应用它们。

例如，对于高性能、事务密集型、企业级应用程序，Groovy 脚本通常不太适合；在这些情况下，您最好的选择 可能是普通的 J2EE 系统。但另一方面，一些脚本 —— 特别是用 Groovy 编写的脚本 —— 会非常有用，因为它能迅速地为小型的、非常特殊的、不是性能密集型的应用程序（例如配置系统/生成系统）快速制作原型。对于报表应用程序来说，Groovy 脚本也是近乎完美的选择，而最重要的是，对单元测试更是如此。

为什么用 **Groovy** 进行单元测试？

是什么让 Groovy 比起其他脚本平台显得更具有吸引力呢？是它与Java 平台无缝的集成。还是因为它是基于 Java 的语言（不像其他语言，是对 JRE 的替代，因此可能基于旧版的处理器），对于 Java 开发人员来说，Groovy 意味着一条短得让人难以置信的学习曲线。而且一旦将这条学习曲线拉直，Groovy 就能提供一个无与伦比的快速开发平台。

从这个角度来说，Groovy 成功的秘密，在于它的语法 就是 Java 语法，但是规则更少。例如，Groovy 不要求使用分号，变量类型和访问修饰符也是可选的。而且，Groovy 利用了标准的 Java 库，这些都是您已经很熟悉的，包括 `Collections` 和 `File/IO` 。而且，您还可以利用任何 Groovy 提供的 Java 库，包括 JUnit。

事实上，令人放松的类 Java 语法、对标准 Java 库的重用以及快捷的生成-运行周期，这些都使 Groovy 成为快速开发单元测试的理想替代品。但是会说的不如会做的，还是让我们在代码中看看它的实际效果！

JUnit 和 **Groovy**

用 Groovy 对 Java 代码进行单元测试简单得不能再简单了，有很多入门选择。最直接的选择是沿袭行业标准 —— JUnit。Unit 的简单性和其功能的强大都是无与伦比的，作为非常有帮助的 Java 开发工具，它的普遍性也是无与伦比的，而且没有什么能够阻挡 JUnit 和 Groovy 结合，所以为什么多此一举呢？实际上，只要您看

过 JUnit 和 Groovy 在一起工作，我敢打赌您就永远再也不会回头！在这里，要记住的关键的事，您在 Java 语言中能用 JUnit 做到的事，在 Groovy 中用 JUnit 也全都能做到；但是需要的代码要少得多。

入门

在您下载了 JUnit 和 Groovy（请参阅 参考资料）之后，您将有两个选择。第一个选择是编写普通的 JUnit 测试用例，就像以前一直做的那样，扩展 JUnit 令人称赞的 `TestCase` 。第二个选择是应用 Groovy 简洁的 `GroovyTestCase` 扩展，它会扩展 JUnit 的 `TestCase` 。第一个选项是您最快的成功途径，它拥有最多与 *Java* 类似的相似性。而另一方面，第二个选择则把您推进了 Groovey 的世界，这个选择有最大的敏捷性。

开始的时候，我们来想像一个 Java 对象，该对象对指定的 `string` 应用了一个过滤器，并根据匹配结果返回 `boolean` 值。该过滤器可以是简单的 `string` 操作，例如 `indexOf()` ，也可以更强大一些，是正则表达式。可能要通过 `setFilter()` 方法在运行时设置将使用的过滤器， `apply()` 方法接受要过滤的 `string` 。清单 1 用普通的 Java 代码显示了这个示例的 `Filter` 接口：

清单 **1.** 一个简单的 **Java Filter** 接口

```
public interface Filter {
  void setFilter(String fltr);
  boolean applyFilter(String value);
}
```

我们的想法是用这个特性从大的列表中过滤掉想要的或者不想要的包名。所以，我建立了两个实现： `RegexPackageFilter` 和 `SimplePackageFilter` 。

把 Groovy 和 JUnit 的强大功能与简单性结合在一起，就形成了如清单 2 所示的简洁的测试套件：

清单 **2.** 用 **JUunit** 制作的 **Groovy RegexFilterTest**

```
import junit.framework.TestCase
import com.vanward.sedona.frmwrk.filter.impl.RegexPackageFilter
class RegexFilterTest extends TestCase {
  void testSimpleRegex() {
    fltr = new RegexPackageFilter()
    fltr.setFilter("java.*")
    val = fltr.applyFilter("java.lang.String")
    assertEquals("value should be true", true, val)
  }
}
```

不管您是否熟悉 Groovy，清单 2 中的代码对您来说应当很面熟，因为它只不过是没有分号、访问修饰符或变量类型的 Java 代码而已！上面的 JUnit 测试中有一个测试用例 `testSimpleRegex()` ，它试图断言 `RegexPackageFilter` 用正则表达式 `"java.*"` 正确地找到了与“ `java.lang.String` ”匹配的对象。

Groovy 扩展了 JUnit

扩展 JUnit 的 `TestCase` 类，加入附加特性，实际上是每个 JUnit 扩展通常采用的技术。例如，DbUnit 框架（请参阅 参考资料）提供了一个方便的 `DatabaseTestCase` 类，能够比以往任何时候都更容易地管理数据库的状态，还有重要的 `MockStrutsTestCase` （来自 StrutsTestCase 框架；请参阅 参考资料），它生成虚拟的 `servlet` 容器，用来执行 struts 代码。这两个强大的框架都极好地扩展了 JUnit，提供了 JUnit 核心代码中所没有的其他特性；而现在 Groovy 来了，它也是这么做的！

与 StrutsTestCase 和 DbUnit 一样，Groovy 对 JUnit 的 `TestCase` 的扩展给您的工具箱带来了一些重要的新特性。这个特殊的扩展允许您通过 `groovy` 命令运行测试套件，而且提供了一套新的 `assert` 方法。可以用这些方法很方便地断言脚本的运行是否正确，以及断言各种数组类型的长度和内容等。

享受 GroovyTestCase 的快乐

了解 `GroovyTestCase` 的能力最好的办法，莫过于实际看到它的效果。在清单 3 中，我已经编写了一个新的 `SimpleFilterTest` ，但是这次我要扩展 `GroovyTestCase` 来实现它：

清单 3. 一个真正的 GroovyTestCase

```
import groovy.util.GroovyTestCase
import com.vanward.sedona.frmwrk.filter.impl.SimplePackageFilter
class SimpleFilterTest extends GroovyTestCase {

  void testSimpleJavaPackage() {
    fltr = new SimplePackageFilter()
    fltr.setFilter("java.")
    val = fltr.applyFilter("java.lang.String")
    assertEquals("value should be true", true, val)
  }
}
```

请注意，可以通过命令行来运行该测试套件，没有运行基于 Java 的 JUnit 测试套件所需要的 `main()` 方法。实际上，如果我用 Java 代码编写上面的 `SimpleFilterTest` ，那么代码看起来会像清单 4 所示的那样：

清单 **4.** 用 **Java** 代码编写的同样的测试用例

```
import junit.framework.TestCase;
import com.vanward.sedona.frmwrk.filter.Filter;
import com.vanward.sedona.frmwrk.filter.impl.SimplePackageFilter;
public class SimplePackageFilterTest extends TestCase {
   public void testSimpleRegex() {
    Filter fltr = new SimplePackageFilter();
    fltr.setFilter("java.");
    boolean val = fltr.applyFilter("java.lang.String");
    assertEquals("value should be true", true, val);
   }

   public static void main(String[] args) {
     junit.textui.TestRunner.run(SimplePackageFilterTest.class);
   }
}
```

用断言进行测试

除了可以让您通过命令行运行测试之外， `GroovyTestCase` 还向您提供了一些特别方便的 `assert` 方法。例如， `assertArrayEquals` ，它可以检查两个数据中对应的每一个值和各自的长度，从而断言这两个数据是否相等。从清单 5 的示例开始，就可以看到 Groovy 断言的实际效果，清单 5 是一个简洁的基于 Java 的方法，它把 `string` 分解成数组。（请注意，我可能使用了 Java 1.4 中新添加的 `string` 特性编写以下的示例类。我采用 Jakarta Commons `StringUtils` 类来确保与 Java 1.3 的后向兼容性。）

清单 **5.** 定义一个 **Java StringSplitter** 类

```
import org.apache.commons.lang.StringUtils;
public class StringSplitter {
  public static String[] split(final String input, final String sep
   return StringUtils.split(input, separator);
  }
}
```

清单 6 展示了用 Groovy 测试套件及其对应的 `assertArrayEquals` 方法对这个类进行测试是多么简单：

清单 **6.** 使用 **GroovyTestCase** 的 **assertArrayEquals** 方法

```
import groovy.util.GroovyTestCase
import com.vanward.resource.string.StringSplitter
class StringSplitTest extends GroovyTestCase {

  void testFullSplit() {
    splitAr = StringSplitter.split("groovy.util.GroovyTestCase", ".
    expect = ["groovy", "util", "GroovyTestCase"].toArray()
    assertArrayEquals(expect, splitAr)
  }
}
```

其他方法

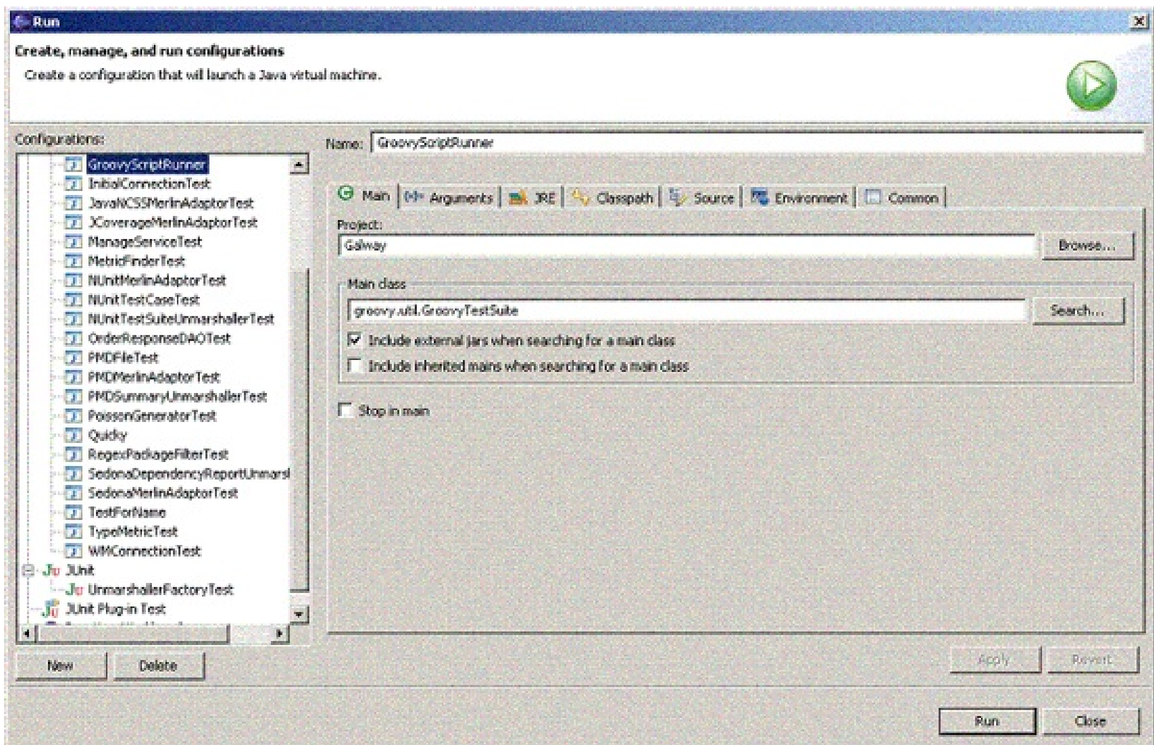
Groovy 可以让您单独或成批运行测试。使用 `GroovyTestCase` 扩展，运行单个测试毫不费力。只要运行 `groovy` 命令，后面跟着要运行的测试套件即可，如清单 7 所示：

清单 **7.** 通过 **groovy** 命令运行 **GroovyTestCase** 测试用例

```
$./groovy test/com/vanward/sedona/frmwrk/filter/impl/SimpleFilterTe
.
Time: 0.047
OK (1 test)
```

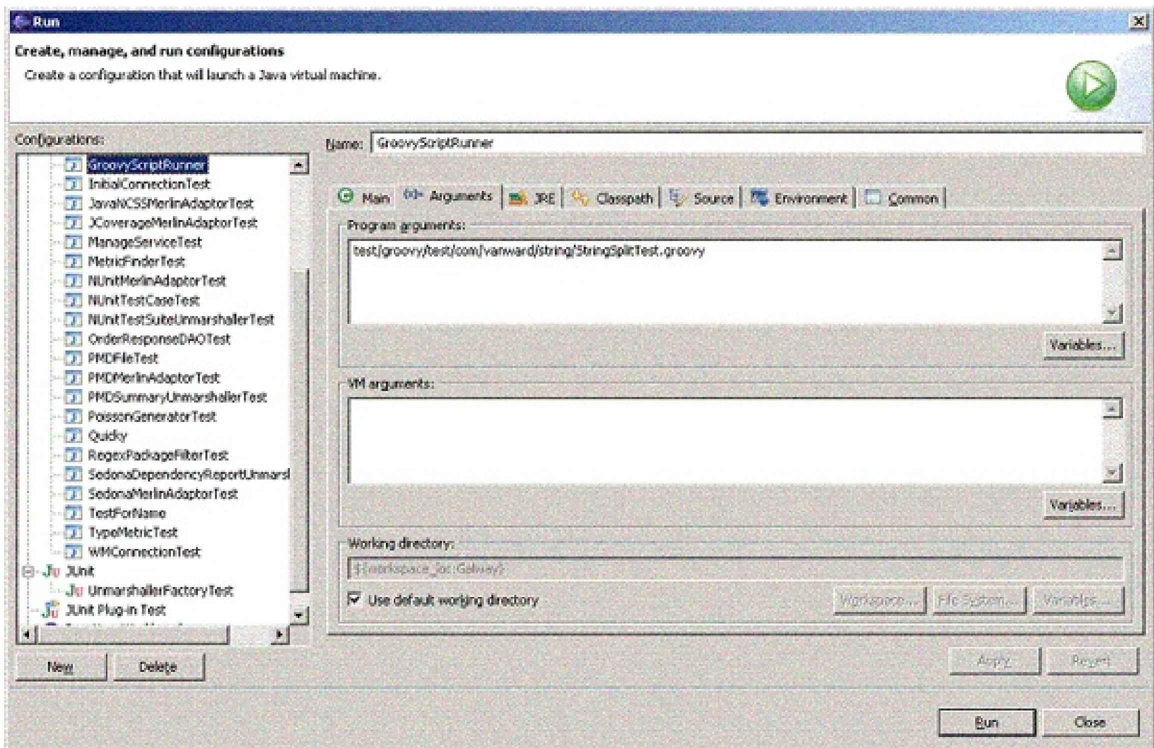
Groovy 还提供了一个标准的 JUnit 测试套件，叫作 `GroovyTestSuite` 。只要运行该测试套件，把脚本的路径传给它，它就会运行脚本，就像 `groovy` 命令一样。这项技术的好处是，它可以让您在 IDE 中运行脚本。例如，在 Eclipse 中，我只是为示例项目建立了一个新的运行配置（一定要选中 “Include external jars when searching for a main class”），然后找到主类 `groovy.util.GroovyTestSuite` ，如图 1 所示：

图 **1.** 用 **Eclipse** 运行 **GroovyTestSuite**



在图 2 中，您可以看到当点击 Arguments 标签，写入脚本的路径时，会发生了什么：

图 **2.** 在 **Eclipse** 中指定脚本的路径



运行一个自己喜欢的 JUnit Groovy 脚本，实在是很简单，只要在 Eclipse 中找到对应的运行配置就可以了。

用 **Ant** 和 **Maven** 进行测试

这个像 JUnit 一样的框架的美妙之处还在于，它可以把整套测试作为 build 的一部分运行，不需要人工进行干预。随着越来越多的人把测试用例加入代码基（code base），整体的测试套件日益增长，形成一个极好的回归平台（regression platform）。更妙的是，Ant 和 Maven 这样的 build 框架已经加入了报告特性，可以归纳 Junit 批处理任务运行的结果。

把一组 Groovy 测试用例整合到某一个构建中的最简单的方法是把它们编译成普通的 Java 字节码，然后把它们包含在 Ant 和 Maven 提供的标准的 Junit 批命令中。幸运的是，Groovy 提供了一个 Ant 标签，能够把未编译的 Groovy 脚本集成到字节码中，这样，把脚本转换成有用的字节码的处理工作就变得再简单不过。例如，如果正在使用 Maven 进行构建工作，那么只需在maven.xml 文件中添加两个新的目标、在 project.xml 中添加两个新的相关性、在 build.properties 文件中添加一个简单的标志就可以了。

我要从更新 maven.xml 文件开始，用新的目标来编译示例脚本，如清单 8 所示：

清单 **8.** 定义 **Groovyc** 目标的新 **maven.xml** 文件

```
<goal name="run-groovyc" prereqs="java:compile,test:compile">

  <path id="groovy.classpath">
    <pathelement path="${maven.build.dest}"/>
    <pathelement path="target/classes"/>
    <pathelement path="target/test-classes"/>
    <path refid="maven.dependency.classpath"/>
  </path>
<taskdef name="groovyc" classname="org.codehaus.groovy.ant.Groovyc
   <classpath refid="groovy.classpath"/>
</taskdef>
<groovyc destdir="${basedir}/target/test-classes" srcdir="${based
         listfiles="true">
   <classpath refid="groovy.classpath"/>
</groovyc>
</goal>
```

上面代码中发生了以下几件事。第一，我定义一个名为 `run-groovyc` 的新目标。该目标有两个前提条件， `java:compile` 编译示例源代码， `test:compile` 编译普通的 Java-JUnit 类。我还用 `<path>` 标签创建了一个 classpath。在该例中，classpath 把 build 目录（保存编译后的源文件）和与它相关的所有依存关系（即 JAR 文件）整合在一起。接着，我还用 `<taskdef>` Ant 标签定义了 `groovyc` 任务。

而且，请您注意我在 classpath 中是如何告诉 Maven 到哪里去找 `org.codehaus.groovy.ant.Groovyc` 这个类。在示例的最后一行，我定义了 `<groovyc>` 标签，它会编译在 `test/groovy` 目录中发现的全部 Groovy

脚本，并把生成的 *.class* 文件放在 `target/test-classes` 目录中。

一些重要细节

为了编译 Groovy 脚本，并运行生成的字节码，我必须要通过 project.xml 文件定义两个新的依存关系（ *groovy* 和 *asm*），如清单 9 所示：

清单 **9. project.xml** 文件中的新的依存关系

```
<dependency>
  <groupId>groovy</groupId>
  <id>groovy</id>
  <version>1.0-beta-6</version>
</dependency>
<dependency>
  <groupId>asm</groupId>
  <id>asm</id>
  <version>1.4.1</version>
</dependency>
```

一旦将脚本编译成普遍的 Java 字节码，那么任何标准的 JUnit 运行器就都能运行它们。因为 Ant 和 Maven 都拥有 JUnit 运行器标签，所以下一步就是让 JUnit 挑选新编译的 Groovy 脚本。而且，因为 Maven 的 JUnit 运行器使用模式匹配来查找要运行的测试套件，所以需要在 build.properties 文件中添加一个特殊标记，如清单 10 所示，该标记告诉 Maven 去搜索类而不是搜索 .java 文件。

清单 **10. Maven** 项目的 **build.properties** 文件

```
maven.test.search.classdir = true
```

最后，我在 maven.xml 文件中定义了一个测试目标（ `goal` ），如清单 11 所示。这样做可以确保 在单元测试运行之前，使用新的 `run-groovyc` 目标编译 Groovy 脚本。

清单 **11. maven.xml** 的新目标

```
<goal name="test">
  <attainGoal name="run-groovyc"/>
  <attainGoal name="test:test"/>
</goal>
```

最后一个，但并非最不重要

有了新定义的两个目标（一个用来编译脚本，另外一个用来运行 Java 和 Groovy 组合而成的单元测试），剩下的事就只有运行它们，检查是不是每件事都顺利运行！

在清单 12 中，您可以看到，当我运行 Maven，给 `test` 传递目标之后，会发生了什么，它首先包含 `run-groovyc` 目标（而该目标恰好还包含 `java:compile` 和 `test:compile` 这两个目标），然后包含 Maven 中自带的标准的 `test:test` 目标。请注意观察目标 `test:test` 是如何处理新生成的 Groovy 脚本（在该例中，是新 编译的 Groovy 脚本） 以及普通的 Java JUnit 测试。

清单 **12.** 运行新的测试目标

```
$ ./maven test
test:
java:compile:
    [echo] Compiling to /home/aglover/dev/target/classes
    [javac] Compiling 15 source files to /home/aglover/dev/target/c
test:compile:
    [javac] Compiling 4 source files to /home/aglover/dev/target/te
run-groovyc:
    [groovyc] Compiling 2 source files to /home/aglover/dev/target/
    [groovyc] /home/aglover/dev/test/groovy/test/RegexFilterTest.gr
    [groovyc] /home/aglover/dev/test/groovy/test/SimpleFilterTest.g
test:test:
    [junit] Running test.RegexFilterTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.6
    [junit] Running test.SimpleFilterTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.6
    [junit] Running test.SimplePackageFilterTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.5
BUILD SUCCESSFUL
Total time: 42 seconds
Finished at: Tue Sep 21 17:37:08 EDT 2004
```

结束语

在 实战 *Groovy* 系列的第一期中，您学习了 Groovy 这个令人兴奋的脚本语言最实用的应用当中的一个。对于越来越多开发人员而言，单元测试是开发过程的重要组成部分；而使用 Groovy 和 JUnit 对 Java 代码进行测试就变成了轻而易举的事情。

Groovy 简单的语法、内部的灵活性，使其成为迅速编写有效的 JUnit 测试、将测试整合到自动编译中的一个优秀平台。对于像我一样为代码质量发狂的人来说，这种组合极大地减少了我的 神经紧张，还让我可以得到我想做得最好的东西：编写“防弹”软件。快点行动吧。

因为这是一个新的系列，所以我非常希望您能一起来推动它前进。如果您对 Groovy 有什么想了解的事情，请 发邮件给我，让我知道您的要求！我希望您会继续支持本系列的下一期，在下期中，我将介绍用 Groovy 进行 Ant 脚本编程。

alt.lang.jre: 感受 Groovy

介绍 *Java* 平台的一种新标准语言

虽然 Java 语言因其严密性和扩展性的承诺而在整整一代程序员中胜出，但是 Groovy 预示了 Java 平台上的一个编程新时代，这种语言是以方便性、适宜性和敏捷性为出发点定义的。在新的 *alt.lang.jre*专栏的第二期文章中，Andrew Glover 对提议添加到 Java 平台的标准编程语言作了非正式的介绍。

如果您在使用 Java 平台（block），不管时间长短，您都有可能听说过 Groovy。Groovy 是超级明星开发人员 James Strachan 和 Bob McWhirter 发明的，它是一种敏捷开发语言，完全以 Java 编程 API 为基础。Groovy 当前正处于 Java Specification Request 的开始阶段，它于 2004 年 3 月底获得批准。Groovy 还是一种脚本语言，有些人说它会永久性地改变您看待和使用 Java 平台的方式。

在其对 JSR 241 （请参阅 参考资料）的开放评论中，Groovy 的共同规范领导者 Richard Monson-Haefel 说他对 Groovy 的支持是建立在总有一天 Java 平台要包括一种敏捷开发语言这一信念上。与许多移植到 Java 平台的脚本语言不同，Groovy 是 _为_JRE 而写的。在规范请求中（参阅 参考资料），Groovy 的制造者提出了“Java 不仅是一种编程语言，更是一个健壮的平台，可以有多种语言在上面运行和共存”（Monson-Haefel 语）的思想。

新 *alt.lang.jre*专栏的这第二期文章的目的是让读者了解 Groovy。我首先回答关于这种新语言的一些最显然的问题（为什么需要它？），然后以代码为基础概述 Groovy 最令人兴奋的功能。

为什么需要另一种语言？

正如在 上个月的专栏中介绍的，Groovy 不是与 JRE 兼容的惟一脚本语言。Python、Ruby 和 Smalltalk 就是成功地移植到 Java 平台上的三种脚本语言。对于一些开发人员，这带来了问题：为什么要另一种语言？毕竟，我们许多人已经将 Java 代码与 Jython 或者 JRuby 结合来快速开发应用程序，为什么还要学习另一种语言？回答是 您不一定要学习一种新语言以用 *Groovy* 编码。Groovy 与其他 JRE 兼容脚本语言的不同在于它的语法以及重用 Java 库。Jython 和 JRuby 共享它们前身（分别是 Python 和 Ruby）的外观，Groovy 让人觉得就像是 Java 语言，不过限制要少得多。

关于本系列

虽然本系列的大多数读者熟悉 Java 语言以及它是如何在跨平台虚拟机上运行的，但是只有少数人知道 Java Runtime Environment 可以承载 Java 语言之外的语言。本系列文章对 JRE 的多种替代语言进行了综述。这里讨论的大多数语言是开放源代码的，许多是免费使用的，有少数是商业产品，必须购买。在 *alt.lang.jre*系列中介绍的所有语言都得到了 JRE 支持，并且作者相信它们增强了 Java 平台的动态性和灵活性特征。

像 Jython 这样的语言是在它们的父语言库上建立的，而 Groovy 使用了 Java 开发人员最熟悉的功能和库 —— 但是将它们放到了一个敏捷开发框架中。敏捷开发的基本宗旨是代码应该很好地适合范围广泛的任务，并可以不同的方式应用。Groovy 通过以下方式落实了这些宗旨：

- 使开发人员不用编译。
- 允许动态类型。
- 使合成结构容易。
- 使其脚本可以在普通 Java 应用程序中使用。
- 提供一个 shell 解析器。

这些特性使 Groovy 成为一种特别容易学习和使用的语言，不管您是有经验的 Java 开发人员还是刚接触 Java 平台。在下面几节中，我将详细讨论上述特性。

看呀，没有 javac！

像许多脚本语言一样，Groovy 不用为运行时编译。这意味着 Groovy 脚本是 在它们运行时解释的，就像 JavaScript 是在观看 Web 页时由浏览器解释的一样。运行时判断会有执行速度的代价，这有可能使脚本语言不能用于对性能有要求的项目，但是无编译的代码在构建-运行周期中可以提供很多好处。运行时编译使 Groovy 成为快速原型设计、构建不同的实用程序和测试框架的理想平台。

脚本的能力

脚本语言很流行，因为它们容易学习并且为开发人员设置的限制较少。脚本语言通常使用简单的、相当简洁的语法，这使开发人员可以用比大多数编程语言所需要的更少的代码创建真实世界的应用程序。像 Perl、Python、Ruby 和现在的 Groovy，用其敏捷方式编写代码而使编程工作达到一个新的效率水平。这种提高的敏捷性通常会使开发人员的效率提高。脚本语言的成功表明脚本不是一种小范围内使用的技术或者黑客的娱乐工具，而是一种由像 Google、Yahoo 和 IBM 这样的世界级公司所使用的切实可行的技术。

例如，运行脚本 Emailer.groovyin Groovy 就是在命令行键入 `groovy Emailer.groovy` 这么容易。如果希望运行同样的 Java 文件（Emailer.java），显然必须键入额外的命令： `javac Emailer.java` ，然后是 `java Emailer` 。虽然这看起来可能有些微不足道，但是可以容易设想运行时编译在应用程序开发的更大上下文中的好处。

可以在稍后看到，Groovy 还允许脚本放弃 main 方法以静态地运行一个关联的应用程序。

动态 dynamo

像其他主流脚本语言一样，Groovy 不需要像 C++ 和 Java 语言这样的正式语言的显式类型。在 Groovy 中，一个对象的类型是在运行时动态发现的，这极大地减少了要编写的代码数量。首先可以通过分析清单 1 和 2 中的简单例子看到这一点。

清单 1 显示了在 Java 语言中如何将一个本地变量声明为 `String` 。注意必须声明类型、名和值。

清单 **1. Java** 静态类型

```
String myStr = "Hello World";
```

在清单 2 中，您看到同样的声明，但是不需要声明变量类型。

清单 **2. Groovy** 动态类型

```
myStr = "Hello World"
```

您可能还注意到了，在清单 2 中我可以去掉声明中的分号。在定义方法及其相关的参数时动态类型有戏剧性的后果：多态具有了全新的意义！事实上，使用动态类型， 不使用继承就可以得到多态的全部能力。在清单 3 中，可以真正开始看到动态类型在 Groovy 的灵活性方面所起的作用。

清单 **3.** 更多 **Groovy** 动态类型

```
class Song{
  length
  name
}
class Book{
  name
  author
}
def doSomething(thing){
  println "going to do something with a thing named = " + thing.nar
}
```

这里，我定义了两个 Groovy 类， `Song` 和 `Book` ，我将在后面对它们进一步讨论。这两个类都包含一个 `name` 属性。我还定义了一个函数 `doSomething` ，它以一个 `thing` 为参数，并试图打印这个对象的 `name` 属性。

因为 `doSomething` 函数没有定义其输入参数的类型，只要对象包含一个 `name` 属性，那么它就可以工作。因此，在清单 4 中，可以看到在使用 `Song` 和 `Book` 的实例作为 `doSomething` 的输入时会有什么现象。

清单 **4.** 试验动态类型

```
mySong = new Song(length:90, name:"Burning Down the House")
myBook = new Book(name:"One Duck Stuck", author:"Phyllis Root")
doSomething(mySong) //prints Burning Down the House
doSomething(myBook) //prints One Duck Stuck
anotherSomething = doSomething
anotherSomething(myBook) //prints One Duck Stuck
```

除了展示 Groovy 中的动态类型，清单 4 的最后两行还揭示了创建对一个函数的引用有多容易。这是因为在 Groovy 中 所有东西都是对象，包括函数。

关于 Groovy 的动态类型声明最后要注意的是，它会导致更少的 `import` 语句。尽管 Groovy 需要 import 以显式使用类型，但是这些 import 可以使用别名以提供更短的名字。

动态类型综述

下面两个例子将到目前为止讨论过的 Groovy 中的动态类型的内容放到一起。下面的 Java 代码组和 Groovy 代码组利用了 Freemarker（参阅 参考资料），这是一个开放源代码模板引擎。这两组代码都只是简单地用一个目录和文件名创建一个 `Template` 对象，然后将相应对象的内容打印到标准输出，当然，不同之处是每一组代码处理这项任务所需要的代码量。

清单 **5.** 简单的 **TemplateReader Java** 类

```
import java.io.File;
import java.io.IOException;
import freemarker.template.Configuration;
import freemarker.template.Template;
public class TemplateReader {
  public static void main(String[] args) {
    try{
      Configuration cfg = Configuration.getDefaultConfiguration();
      cfg.setDirectoryForTemplateLoading(
           new File("C:\\dev\\projects\\http-tester\\src\\conf"));

      Template temp = cfg.getTemplate("vendor-request.tmpl");

        System.out.println(temp.toString());
      }catch(IOException e){
        e.printStackTrace();
      }
  }
}
```

初看之下，清单 5 中的 Java 代码相当简单 —— 特别是如果以前从来没见过脚本代码时。幸运的是，有清单 6 中的 Groovy 作为对比。现在这段代码很简单！

清单 **6.** 用 **Groovy** 编写的更简单的 **TemplateReader**

```
import freemarker.template.Configuration as tconf
import java.io.File
cfg = tconf.getDefaultConfiguration()
cfg.setDirectoryForTemplateLoading(
  new File("C:\\dev\\projects\\http-tester\\src\\conf"))

temp = cfg.getTemplate("vendor-request.tmpl")
println temp.toString()
```

Groovy 代码只有 Java 代码的一半那么长，下面是原因：

- Groovy 代码只需要一半的 `import` 语句。还要注意，`freemarker.template.Configuration` 使用了别名 `tconf` ，使得语法更短。
- Groovy 允许类型为 `Template` 的变量 `tmpl` 不声明其类型。
- Groovy 不需要 `class` 声明或者 `main` 方法。
- Groovy 不关心任何相应异常，使您可以不用导入 Java 代码中需要的 `IOException` 。

现在，在继续之前，想一下您所编写的最后一个 Java 类。您可能不得不编写很多 import 并声明类型，并在后面加上同样数量的分号。考虑用 Groovy 编写同样的代码会是什么情况。可以使用简练得多的语法，不需要遵守这么多的规则，并且得到完全相同的行为。

想一下，如果您正好是刚刚开始……

特别灵活的语法

谈到语法，灵活性是更有效地开发代码的主要因素。很像其有影响的对手（Python、Ruby 和 Smalltalk），Groovy 极大地简化了核心库的使用和它所模拟的语言（在这里是 Java 语言）的构造。为了让您对 Groovy 语法的灵活性有一个大体概念，我将展示它的一些主要结构，即类、函数（通过 `def` 关键词）、闭包、集合、范围、映射和迭代器。

类

在字节码水平，Groovy 类是眞正的 Java 类。不同之处在于 Groovy 将类中定义的所有内容都默认为 `public` ，除非定义了特定的访问修饰符。而且，动态类型应用到字段和方法，不需要 `return` 语句。

在清单 7 中可以看到 Groovy 中类定义的例子，其中类 `Dog` 有一个 `getFullName` 方法，它实际上返回一个表示 `Dog` 的全名的 `String` 。并且所有方法都隐式地为 `public` 。

清单 **7.** 示例 **Groovy** 类：**Dog**

```
class Dog{
  name
  bark(){
    println "RUFF! RUFF!"
  }

  getFullName(master){
    name + " " + master.lname
  }

  obeyMaster(){
    println "I hear you and will not obey."
  }
}
```

在清单 8 中，推广到有两个属性 —— `fname` 和 `lname` —— 的类 `DogOwner`，就是这么简单！

清单 **8.** 示例 **Groovy** 类：**DogOwner**

```
class DogOwner{
  fname
  lname
  trainPet(pet){
    pet.obeyMaster()
  }

}
```

在清单 9 中，用 Groovy 设置属性并对 `Dog` 和 `DogOwner` 实例调用方法。现在很明显，使用 Groovy 类比 Java 类要容易得多。虽然需要 `new` 关键词，但是类型是可选的，且设置属性（它隐式为 public）是相当轻松的。

清单 **9.** 使用 **Groovy** 类

```
myDog = new Dog()
myDog.name = "Mollie"
myDog.bark()
myDog.obeyMaster()
me = new DogOwner()
me.fname = "Ralf"
me.lname = "Waldo"
me.trainPet(myDog)
str = myDog.getFullName(me)
println str  // prints Mollie Waldo
```

注意在 `Dog` 类中定义的 `getFullName` 方法返回一个 `String` 对象，在这里它是“ `Mollie Waldo` ”。

第一类对象

第一类对象 是可以在运行时用数据创建并使用的对象。第一类对象还可以传递给函数和由函数输出、作为变量存储、由其他对象返回。Java 语言自带的基本数据类型，如 `int` 和 `boolean` ，不认为是第一类对象。许多面向对象纯粹论者哀叹这个小细节，一些人据此提出 Java 语言是否是真正的面向对象语言。Groovy 通过将所有内容声明为对象而解决了这一问题。

Def

除了像许多脚本语言那样将所有对象指定为第一类对象（见侧栏），Groovy 还让您创建 第一类函数，它本身实质上就是对象。它们是用 `def` 关键词定义的并在类定义之外。您实际上在 清单 3 中已经看到了如何用 `def` 关键词定义第一类函数，并在 清单 4中看到使用了一个函数。Groovy 的第一类函数定义简单脚本时特别有用。

闭包

Groovy 中最令人兴奋和最强大的功能是支持闭包。 闭包（*Closure*）是第一类对象，它类似于 Java 语言中的匿名内部类。闭包和匿名内部类都是可执行的一段代码，不过这两者之间有一些细微的不同。状态是自动传入传出闭包的。闭包可以有名字。它们可以重复使用。而且，最重要且对 Groovy 同样成立的是，闭包远比匿名内部类要灵活得多！

清单 10 展示了闭包的强大。清单中新的和改进的 `Dog` 类包括一个 `train` 方法，它实际上执行创建了 `Dog` 实例的闭包。

清单 **10.** 使用闭包

```
class Dog{
  action
  train(){
    action.call()
  }
}
sit = { println "Sit, Sit! Sit! Good dog"}
down = { println "Down! DOWN!" }
myDog = new Dog(action:sit)
myDog.train()  // prints Sit, Sit! Sit! Good dog
mollie = new Dog(action:down)
mollie.train() // prints Down! DOWN!
```

而且，闭包还可以接收参数。如清单 11 所示， `postRequest` 闭包接收两个参数（ `location` 和 `xml` ），并使用 Jakarta Commons HttpClient 库（参阅 参考资料）将一个 XML 文档发送给指定位置。然后这个闭包返回一个表示响应的 `String` 。闭包定义下面是一个使用闭包的例子。事实上，调用闭包就像调用函数一样。

清单 **11.** 使用带参数的闭包

```
import org.apache.commons.httpclient.HttpClient
import org.apache.commons.httpclient.methods.PostMethod
postRequest = { location, xml |
  clint = new HttpClient()
  mthd = new PostMethod(location)
  mthd.setRequestBody(xml)
  mthd.setRequestContentLength(xml.length())
  mthd.setRequestHeader("Content-type",
     "text/xml; charset=ISO-8859-1")

  statusCode = clint.executeMethod(mthd)
  responseBody = mthd.getResponseBody()
  mthd.releaseConnection()
  return new String(responseBody)
}
loc = "http://localhost:8080/simulator/AcceptServlet/"
vxml = "<test><data>blah blah blah</data></test>"
str = postRequest(loc, vxml)
println str
```

自动装箱

自动装箱或者装箱转换是一个自动将像 `int` 、 `double` 和 `boolean` 这样的基本数据类型自动转换为可以在 `java.lang` 包中找到的它们的相应包装类型的过程。这一功能出现在 J2SE 1.5 中，使开发人员不必在源代码中手工编写转换代码。

集合

将对象组织到像列表和映射这样的数据结构中是一项基本的编码任务，是我们大多数人每天要做的工作。像大多数语言一样，Groovy 定义了一个丰富的库以管理这些类型的集合。如果曾经涉足 Python 或者 Ruby，那么应该熟悉 Groovy 的集合语法。如清单 12 所示，创建一个列表与在 Java 语言中创建一个数组很类似。（注意，列表的第二项自动装箱为一个 `Integer` 类型。）

清单 **12.** 使用集合

```
collect = ['groovy', 29, 'here', 'groovy']
```

除了使列表更容易处理，Groovy 还为集合增加了几个新方法。这些方法使得，如 `统计` 值出现的次数、将整个列表 `结合` 到一起、对列表 `排序` 变得非常容易。可以在清单 13 中看到这些集合方法的使用。

清单 **13.** 使用 **Groovy** 集合

```
aCollect = [5, 9, 2, 2, 4, 5, 6]
println aCollect.join(' - ')  // prints 5 - 9 - 2 - 2 - 4 - 5 - 6
println aCollect.count(2)     // prints 2
println aCollect.sort()       // prints [2, 2, 4, 5, 5, 6, 9]
```

Maps

像列表一样，映射也是一种在 Groovy 中非常容易处理的数据结构。清单 14 中的映射包含两个对象，键是 `name` 和 `date` 。注意可以用不同的方式取得值。

清单 **14.** 处理映射

```
myMap = ["name" : "Groovy", "date" : new Date()]
println myMap["date"]
println myMap.date
```

范围

在处理集合时，很可能会大量使用 `范围（Range）` 。 `范围` 实际上是一个很直观的概念，并且容易理解，利用它可以包含地或者排除地创建一组有序值。使用两个点（ `..` ）声明一个包含范围，用三个点（ `...` ）声明一个排除范围，如清单 15 所示。

清单 **15.** 处理范围

```
myRange = 29...32
myInclusiveRange = 2..5
println myRange.size() // prints 3
println myRange[0]   // prints 29
println myRange.contains(32) //prints false
println myInclusiveRange.contains(5) //prints true
```

用范围实现循环

在循环结构中，范围可以实现相当巧妙的想法。在清单 16 中，将 `aRange` 定义为一个排除范围，循环打印 a、b、c 和 d。

清单 **16.** 用范围实现循环

```
aRange = 'a'...'e'
for (i in aRange){
  println i
}
```

集合的其他功能

如果不熟悉 Python 和其他脚本语言，那么您在 Groovy 集合中发现的一些其他功能会让您印象深刻。例如，创建了集合后，可以用负数在列表中反向计数，如清单 17 所示。

清单 **17.** 负索引

```
aList = ['python', 'ruby', 'groovy']
println aList[-1] // prints groovy
println aList[-3] // prints python
```

Groovy 还让您可以用范围分割列表。分割可获得列表的准确子集，如清单 18 所示。

清单 **18.** 用范围分割

```
fullName = "Andrew James Glover"
mName = fullName[7...13]
print "middle name: " + mName // prints James
```

集合类似于 **Ruby**

如果愿意的话，还可以将 Groovy 集合作为 Ruby 集合。可以用类似 Ruby 的语法，以 `<<` 语法附加元素、用 `+` 串接和用 `-` 对集合取差，甚至还可以用 `*` 语法处理集合的重复，如清单 19 所示。 注意，还可以用 `==` 比较集合。

清单 **19. Ruby** 风格的集合

```
collec = [1, 2, 3, 4, 5]
collec << 6 //appended 6 to collec
acol = ['a','b','c'] * 3 //acol now has 9 elements
coll =  [10, 11]
coll2 = [12, 13]
coll3 = coll + coll2 //10,11,12,13
difCol = [1,2,3] - [1,2] //difCol is 3
assert [1, 2, 3] == [1, 2, 3] //true
```

迭代器

在 Groovy 中，迭代任何序列都相当容易。迭代字符序列所需要的就是一个简单的 `for` 循环，如清单 20 所示。（正如您现在可能注意到的，Groovy 提供了比 Java 1.5 以前的传统语法更自然的 `for` 循环语法。）

清单 **20.** 迭代器示例

```
str = "uncle man, uncle man"
for (ch in str){
  println ch
}
```

Groovy 中的大多数对象具有像 `each` 和 `find` 这样的以闭包为参数的方法。用闭包来迭代对象会产生几种令人兴奋的可能性，如清单 21 所示。

清单 **21.** 带有迭代器的闭包

```
[1, 2, 3].each {
  val = it
  val += val
  println val
}
[2, 4, 6, 8, 3].find { x |
     if (x == 3){
       println x
     }
}
```

在清单 21 中，方法 `each` 作为迭代器。在这里，闭包添加元素的值，完成时 `val` 的值为 6。 `find` 方法也是相当简单的。每一次迭代传递进元素。在这里，只是测试值是否为 3。

Groovy 的高级内容

到目前为止，我着重讲述的都是使用 Groovy 的基本方面，但是这种语言有比基本内容多得多的内容！我将以分析 Groovy 提供的一些高级开发功能作为结束，包括 Groovy 样式的 JavaBeans 组件、文件 IO、正则表达式和用 `groovyc` 编译。

GroovyBean！

永远不变的是，应用程序最后要使用类似 struct 的对象表示真实世界的实体。在 Java 平台上，称这些对象为 JavaBean 组件，它们通常用于表示订单、客户、资源等的业务对象。Groovy 由于其方便的简写语法，以及在定义了所需 bean 的属性后

自动提供构造函数，而简化了 JavaBean 组件的编写。结果自然就是极大地减少了代码，正如您可以自己从清单 22 和 23 中看到的。

在清单 22 中，可看到一个简单的 JavaBean 组件，它是用 Java 语言定义的。

清单 **22.** 一个简单的 **JavaBean** 组件

```
public class LavaLamp {
  private Long model;
  private String baseColor;
  private String liquidColor;
  private String lavaColor;
  public String getBaseColor() {
    return baseColor;
  }
  public void setBaseColor(String baseColor) {
    this.baseColor = baseColor;
  }
  public String getLavaColor() {
    return lavaColor;
  }
  public void setLavaColor(String lavaColor) {
    this.lavaColor = lavaColor;
  }
  public String getLiquidColor() {
    return liquidColor;
  }
  public void setLiquidColor(String liquidColor) {
    this.liquidColor = liquidColor;
  }
  public Long getModel() {
    return model;
  }
  public void setModel(Long model) {
    this.model = model;
  }
}
```

在清单 23 中，可以看到用 Groovy 写这个 bean 时所发生的事。所要做的就是定义属性，Groovy 会自动给您一个很好的构造函数以供使用。Groovy 还使您在操纵 `LavaLamp` 的实例时有相当大的灵活性。例如，我们可以使用 Groovy 的简写语法或者传统的冗长的 Java 语言语法操纵 bean 的属性。

清单 **23.** 用 **Groovy** 编写的 **JavaBeans** 组件

```
class LavaLamp{
  model
  baseColor
  liquidColor
  lavaColor
}
llamp = new LavaLamp(model:1341, baseColor:"Black",
  liquidColor:"Clear", lavaColor:"Green")
println llamp.baseColor
println "Lava Lamp model ${llamp.model}"
myLamp = new LavaLamp()
myLamp.baseColor = "Silver"
myLamp.setLavaColor("Red")
println "My Lamp has a ${myLamp.baseColor} base"
println "My Lava is " + myLamp.getLavaColor()
```

轻松的 IO

Groovy IO 操作很轻松，特别是与迭代器和闭包结合时。Groovy 使用标准 Java 对象如 `File` 、 `Reader` 和 `Writer` ，并用接收闭包作参数的额外方法增强了它们。如在清单 24 中，可以看到传统的 `java.io.File` ，但是带有额外的、方便的 `eachLine` 方法。

清单 24. Groovy IO

```
import java.io.File
new File("File-IO-Example.txt").eachLine{ line |
 println "read the following line -> " + line
}
```

因为文件实质上是一系列行、字符等，所以可以相当简单地迭代它们。 `eachLine` 方法接收一个闭包并迭代文件的每一行，在这里是 `File-IO-Example.txt` 。 以这种方式使用闭包是相当强大的，因为 Groovy 保证所有文件资源都是关闭的，不考虑任何异常。这意味着无需大量 `try` / `catch` / `finally` 子句就可以进行文件 IO！

高级编译

Groovy 脚本实际上是字节码级别的 Java 类。因此，可以容易地用 `groovyc` 编译 Groovy 脚本。可以通过命令行或者 `Ant` 使用 `groovyc` 以生成脚本的类文件。这些类可以用普通 `java` 命令运行，只要 classpath 包括 `groovy.jar` 和 `asm.jar` ，这是 ObjectWeb 的字节码操纵框架。要了解更多编译 Groovy 的内容，请参阅 参考资料。

最大 RegEx

如果一种语言没有正则表达式处理，则它是没价值的。Groovy 使用 Java 平台的 `java.util.regex` 库 —— 但是做了少量基本的改变。例如，Groovy 使您可以用 `~` 表达式创建 `Pattern` 对象，用 `=~` 表达式创建 `Matcher` 对象，如清单 25 所示。

清单 **25. Groovy RegEx**

```
str =  "Water, water, every where,
        And all the boards did shrink;
        Water, water, every where,
        Nor any drop to drink."
if (str =~ 'water'){
  println 'found a match'
}
ptrn = ~"every where"
nStr = (str =~ 'every where').replaceAll('nowhere')
println nStr
```

您可能已经注意到了，可以在上述清单中定义 `String` 、 `str` ，而无需为每一新行添加结束引号和 `+` 。这是因为 Groovy 放松了要求字符串串接的普通 Java 约束。运行这段 Groovy 脚本会对匹配 `water` 的情况打印出 `true` ，然后打印出一节诗，其中所有出现“ `every where` ”的地方都替换为“ `nowhere` ”。

关于（**band**）**shell**

Groovy 提供了两种不同的解释器，使所有有效的 Groovy 表达式可以交互地执行。这些 shell 是特别强大的机制，可以用它们迅速学习 Groovy。

结束语

像所有婴儿期的项目一样，Groovy 是正在发展的语言。习惯于使用 Ruby 和 Python（或者 Jython）的开发人员可能会怀念 mixins、脚本导入（尽管可以将所需要的可导入脚本编译为相应的 Java 类）和方法调用的命名参数等这些功能的方便性。 但是 Groovy 绝对是一种发展中的语言。随着其开发人员数量的增加，它很有可能结合这些功能及更多功能。

同时，Groovy 有很多优点。它很好地融合了 Ruby、Python 和 Smalltalk 的一些最有用的功能，同时保留了基于 Java 语言的核心语法。对于熟悉 Java 平台的开发人员，Groovy 提供了更简单的替代语言，且几乎不需要学习时间。对于刚接触 Java 平台的开发人员，它可以作为有更复杂语法和要求的 Java 语言的一个容易的入口点。

像在本系统讨论的其他语言一样，Groovy 不是要替代 Java 语言，而是作为它的另一种选择。与这里讨论的其他语言不一样，Groovy 遵循 Java 规范，这意味着它有可能与 Java 平台上的 Java 语言具有同等重要的作用。

在本月这一期 *alt.lang.jre* 文章中，介绍了 Groovy 的基本框架和语法，以及它的一些高级编程功能。下个月将介绍在 Java 开发人员中最受欢迎的脚本语言：JRuby。